Charles University in Prague
Faculty of Mathematics and Physics

Master Thesis

Josef Hala

Real-life Middleware Support for Connectors

Department of Software Engineering
Supervisor: RNDr. Tomáš Bureš, Ph.D.
Study Program: Computer Science

I would like to thank my advisor Tomáš Bureš, I very appreciate his valuable suggestions and help with writing.

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on April 20, 2007 Josef Hala

Název práce: Real-life middleware support for connectors
Autor: Josef Hala
Katedra: Katedra Softwarového Inženýrství
Vedoucí diplomové práce: RNDr. Tomáš Bureš, Ph.D.
e-mail vedoucího: bures@nenya.ms.mff.cuni.cz

Abstrakt: *Softwarové konektory se používají v komponentových systémech pro realizaci mezi-komponentové komunikace. Pro jejich generovaní slouží generátor konektorů, který z konfiguračního souboru umí vytvořit odpovídajíci konektor. Během návrhu aplikace generátor konektorů umožňuje specifikovat, jak mají jednotlivé komponenty mezi sebou kooperovat. Za běhu aplikace pak vytvořené konektory zajištují komunikaci, jak bylo uvedeno v konfiguraci. Konektory k tomu typicky využívají nějaký midleware a můžou tak být považovany za abstrakci, která zakrývá rozdíly mezi jednotlivými middlewary.*

Cílem této diplomové práce je rozšířit existující generátor konektorů [1] a poskytnout podporu pro generování konektorů využívájících technologie RMI a CORBA a také jiné komunikační styly (především messaging). Kromě implementace různých middlewarů práce rozebírá pokročilejší témata, jako předávání referencí a vypořádání se s komplexními typy v rámci vzdálené komunikace.

Klíčová slova: softwarové konektory, komponentové aplikace, komunikační middleware, framework pro middleware

Title: Real-life middleware support for connectors
Author: Josef Hala
Department: Department of Software Engineering
Supervisor: RNDr. Tomáš Bureš, Ph.D.
Supervisor's e-mail address: bures@nenya.ms.mff.cuni.cz

Abstract: *Software connectors are used in component-based systems for realization of inter-component communication. A connector generator serves for their generation. It is able to create particular connector based on definitions in a high-level configuration file. At design time the connector generator allows for specifying how components interoperate, at run time created connectors are responsible for communication between application's components as it was specified in the configuration file. They typically utilize some middleware for realization of the communication and they can be supposed as a higher level of abstraction which covers the differences between various underlying middlewares.*

The aim of the thesis is to extend the existing connector generator [1] and provide support for generating and deploying RMI and CORBA-based connectors and also connectors using other communication styles (particularly messaging). Besides the actual implementation of various middlewares in the connector generator the thesis also addresses some advanced topics like passing references and handling of complex types within remote communication.

Keywords: software connectors, component-based applications, communication middleware, framework for middleware

Table of Contents

Chapter 1

Introduction

1.1 Software components and connectors

There are many approaches and paradigms to software development. One of them is the so-called Component-based Software Engineering (CBSE) [2]. This is a discipline which emphasizes on splitting a large system into small logical, or functional, components. These components can interact with each other only through well-defined interfaces. Any other communication is forbidden which makes for good encapsulation. Components can be regarded as a higher abstraction of objects – they are more separate, they do not share nor expose any internal data or state, and from the outside they are accessible strictly via interfaces. There is also another difference compared to objects - while a component states explicitly which interfaces it offers, it can also require some interfaces.

Clemens Szyperski [3] and David Messerschmitt [4] give these five criteria that component systems must satisfy (for more information see *Software componentry* [5]):

- Multiple-use
- Non-context-specific
- Composable with other components
- Encapsulated i.e., non-investigable through its interfaces
- A unit of independent deployment and versioning

These features of component systems help by reducing system's complexity and increasing it's reusability, maintainability and scalability. A simple component-based application is depicted in Figure 1.1.

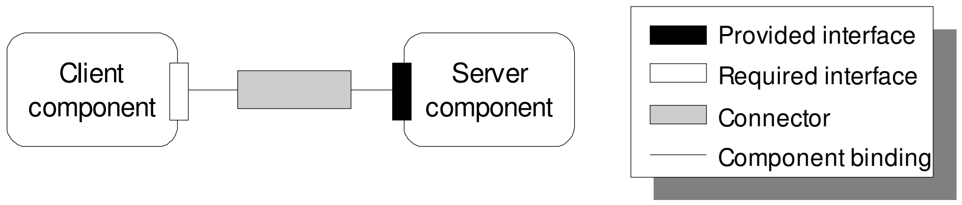


Figure 1.1: A simple component application

Components also have a connection with an Interface description language (IDL), a computer language used to describe a component's interface so that it should be language-neutral in order that components written in different programming languages (as well as running on different operating systems) can interoperate with each other. IDL acts like a “bridge” between these distinct parts.

There are many different component systems developed by various companies and groups. The most well-known of these include COM, DCOM and the recent .NET component model developed by Microsoft Corporation [6], Enterprise JavaBean (EJB) [7] by Sun Microsystems, and the CORBA Component Model (CCM) [8][9] proposed by the Object Management Group (OMG) [10]. Besides these industrial technologies we should also mention SOFA [11] project (being developed at Charles University in Prague), and the modular Fractal Project [12].

A component is a software entity that implements some business logic. Programmers should not have to bothered with low-level communication, they should concentrate on the implementation encapsulated in the component. It should not matter which middleware is used, the communication API should be unified and generalized. For this purpose *software connectors* are highly suitable.

A software connector is a part of an application which takes care of the communication between components, it is supposed to be a higher abstraction of component's interoperability. It can be implemented in various ways and also be aimed at slightly different areas, but its main purpose is dealing with low-level middleware, to connect and bridge differences between heterogeneous components For the user a software connector should act as if it were one uniform entity with a common API.

Besides the invaluable help in hiding component and communication dissimilarities software connectors can be also useful in other ways:

- Connectors can provide facilities for describing the properties of communication. At deployment time they can serve as a modeling tool through which a programmer can specify characteristics and an architecture of the connection.
- At runtime they can add some extra values to the connection itself, e.g. logging, measurement, encryption, increasing safety, reliability, etc.

In order that the connectors are actually useful in practice, they must be easy to create. Therefore there should be a *connector generator* which is capable of creating, from human readable configuration files, connectors both suitable and effective for the particular application.

The connector generator is a matter of deployment compared with connectors which are utilized at runtime. They should not be confused, the connector generator generates connectors.

For example Figure 1.1 shows an application consisting of only two components – client and server. The components say which interfaces they provide (and require) and on the basis of this information the connector generator should generate a specific connector for that application.

1.2 Goals

The goal of this thesis is to continue the work on the existing connector generator [1] which is being developed at Charles University in Prague. It has a modular and very extensible architecture and so that has the possibility to support many different middlewares. At the moment it is only the framework that allows the possibility, but does not yet actually provide the functionality. The challenge is to utilize selected middlewares and integrate them into this system, a considerable amount of work in an unresearched area. Besides the actual implementation of various middlewares in the connector generator we will also have to deal with some advanced topics like passing references and handling of complex types within remote communication.

Chapter 2

Middleware

"Middleware is connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network. Middleware is essential to migrating mainframe applications to client/server applications and to providing for communication across heterogeneous platforms."

Bray M.: Middleware [13]

Connectors use middleware to handle low level communication between software components. The connector generator is highly configurable and contains facilities for integration and support for generating connectors using various middlewares.

We have chosen the following three middleware families because they cover the most common situations and needs of a component's programmers: Java Remote Method Invocation (RMI), CORBA (Common Object Request Broker Architecture), and the Java Message Service (JMS). They have been chosen as they are well known, very popular and have very excellent support in the Java language. Here are brief descriptions of the three.

2.1 Java Remote Method Invocation (RMI)

The *Java Remote Method Invocation* (RMI) API [14] allows a Java program running in a Java Virtual Machine (JVM) [15] to invoke methods from another Java Virtual Machine (which can be on the same host or on a remote host). The main concept is that the user does not have to know that he is calling a remote method, it should behave like a local one, and be transparent to the user. Although there are some limitations (especially when passing complex objects as arguments) RMI is quite easy to use and in simple cases the user really does not have to be concerned with the fact that calls are remote.

There are two main implementations of the RMI API. The first, and original, one uses serialization[1] for transmitting an object across the network. This is unique to Java and can therefore only be used when communicating between Java programs, and disallows Communication with components written in other programming languages. The "background" protocol used in this implementation is the Java

1 Serialization [16] is the process of saving an object onto a storage medium (such as a file, or a memory buffer)

Remote Method Protocol (JRMP) [17].

The term RMI is used to denote just the programming interface or both the API and JRMP.

A typical communication between a server and a client is shown in Figure 2.1.

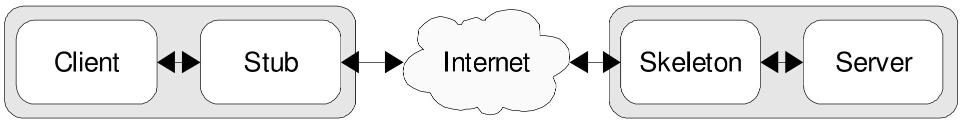


Figure 2.1: A typical implementation of Java RMI. Stub and Skeleton are are special intermediate objects.

The `Stub` and the `Skeleton` form the interlink between the server and the client, they translate function calls into a protocol which is transportable over TCP/IP.

The second RMI implementation is known as RMI-IIOP [18] (Remote Method Invocation over Internet Inter-ORB Protocol). This RMI delegates most functionality to the supporting CORBA implementation, using the low-level IIOP protocol to allow cooperation with CORBA components.

RMI-IIOP is based on open standards defined by the Object Management Group (OMG) [19] and through its openness there are many participating vendors and companies. The main strength of RMI-IIOP is that we can combine Java components with components written in non-Java language (for example in Java, C++, Python, Smalltalk and Lisp).

Here are some alternatives to the built-in Java RMI:

- LipeRMI [20] - is a completely new RMI implementation to replace native Java RMI. It is totally independent from the native Java RMI and uses an Internet optimized approach for the communication layer.
- cajo [21] - provides an easy to use framework to simplify the use of RMI.

2.2 CORBA

CORBA (Common Object Request Broker Architecture) [22] is a standard for distributed computing. It enables software components written in multiple programming languages to interoperate. CORBA uses an *Interface Definition Language* (IDL) for the description of remote interfaces. Here is an example of a simple IDL:

```
interface Car {
    void setSpeed(in long new_speed);
};
```

Figure 2.2: Example of a simple IDL

IDL is a common computer language for defining interfaces and data types. From the IDL code, which is general and system independent, language-specific code is generated. OMG specifies mapping from IDL into C, C++, Python, Ada, Java, Lisp, Smalltalk and PL/I.

Figure 2.2 (above) shows the definition of a simple interface `Car` with one function `setSpeed()`. The corresponding Java interface produced from this IDL would be as shown in Figure 2.3:

```
public interface Car
{
  void setSpeed (int new_speed);
}
```

Figure 2.3: A corresponding Java interface generated from the IDL in Figure 12

Typically each language which the OMG defines mapping for has some utility that generates source code from an IDL into its own language.

For example Java has a command line tool `idlj` which, besides the interface Java code itself, generates also auxiliary files, in particular the stub and skeleton (see later). The generated files contain Java source, which then have to be compiled into binary code using `javac` or some other Java compiler. This process is shown in Figure 2.4:

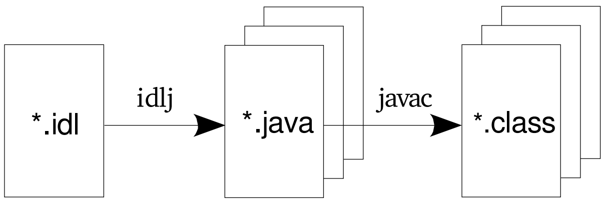


Figure 2.4: Generation of Java bindings from IDL

As mentioned some generated classes are used in the stub and skeleton. Stub and skeleton, which are a form of deletage. For example, on the client side a remote reference is in fact a reference to the stub. The client uses it as a normal object, but all method invocations on the stub are forwarded to an ORB and finally to the server, see Figure 2.5:

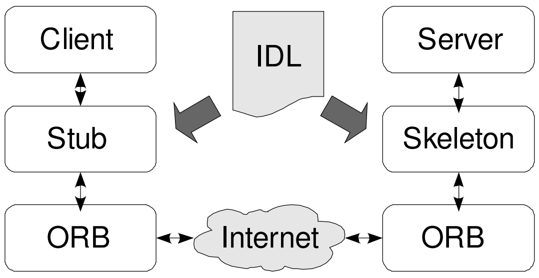


Figure 2.5: A simplified architecture of a typical CORBA application

The client's call to the stub is translated and modified so that ORB can transport it. On the server side the skeleton is doing a reverse translation and the corresponding server's method is called. The return value is transmitted back through the skeleton, decoded by the stub, and finally passed back to the client.

The picture above is very simplified and does not show the *POA* (Portable Object Adapter) which is normally used. The POA is a CORBA object which is responsible for delivering the remote invocation to the correct servant[1] [23][24]. The POA is very flexible and robust. It can call either the local object or redirect the call to some other server and in addition the servant can be chosen statically (once) or dynamically (for each remote invocation).

The *ORB* (Object Request Broker) [24] performs the physical communication between hosts. It handles a transformation of data structures from the host specific format to a byte sequence which can be then transmitted over the network. This process is called marshaling. In addition to marshaling, the ORB often provides other features like transaction or security services.

In order that different ORBs can communicate with each other there exists an abstract protocol, the *General InterORB Protocol* (GIOP), from which concrete protocols are derived, such as the Internet Inter-Orb Protocol (IIOP) [25] for use over TCP/IP layer, and the SSL InterORB Protocol (SSLIOP) which adds encryption and authentication.

The IIOP protocol is also used in RMI-IIOP (see Chapter 7.1.2).

Here is a list of some concrete CORBA implementations:

- JacORB - A Free Software (LGPL) ORB implemented in Java.
- OpenORB - A Free Software (BSD) ORB for Java.
- IIOP.NET - A Free Software (LGPL) ORB for Microsoft .NET.
- omniORB - A Free Software (LGPL) ORB for C++ and Python.
- Borland Enterprise Server, VisiBroker Ed. - A CORBA 2.6–compliant commercial ORB for Java and C++ from Borland.
- BEA Tuxedo - A CORBA 2.5–compliant commercial ORB for Java and C++

1 Servant (associated with POA) is an entity which at the end processes a method call, it contains an implementation of some interface.

from BEA Systems.

2.3 JMS

The Java Message Service (JMS) API [26] brings support for messaging into the Java language.

Messaging is a kind of communication between software components. It is comparable to e-mail when “clients” can send and receive messages to and from each other.

The sending and receiving components need not be running at the same time. It is only important to use the correct destination. One can send message without knowing anything about the receiver. Similarly the receiver of the message need not to know who has created it.

The JMS API provides facilities for creating, sending and receiving messages. Two approaches to messaging exists:

- **Point-to-point Messaging**

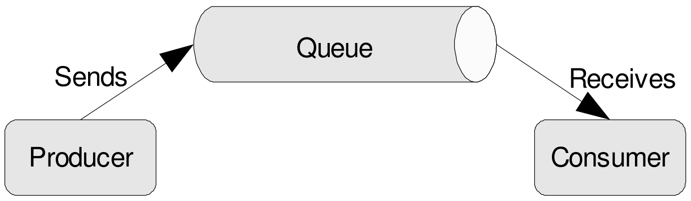


Figure 2.6: Point-to-point messaging

  Point-to-point messaging has producers, queues and consumers. Producers send messages into a specific queue and consumers then receive the messages from it. The important fact is that the queue holds messages until they are received by a client, or until they expire. Each message has only one consumer. When a client receives a message it is removed from the queue.

- **Publish/Subscribe Messaging**

  In publish/subscribe messaging there are producers (which publish messages), consumers (which receive them) and topics. A topic distributes messages from producers (publishers) to all consumers which are subscribed to the topic. Each message can be received by many clients (or by no clients if there are no subscribers existing when the message is published). Another difference compared to PTP (point-to-point) messaging is a time dependency - consumers only receive messages AFTER their subscription, messages published before subscription are not delivered.

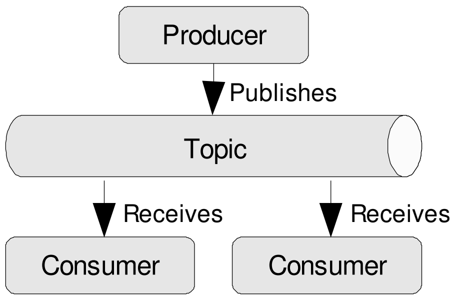


Figure 2.7: Publish/Subscribe messaging

Here are some concrete JMS implementations:

- Java EE Platform – the JMS API is implemented directly in the platform since release 1.3.
- Apache ActiveMQ [27] - a popular and powerful open source Message Broker.
- OpenJMS [28] - an open source implementation of the JMS API.

Chapter 3

Connectors

The main purpose of software connectors is to connect distinct components. They act as a mediator which completely cares about a physical communication. As has been said components should aim on the solution of a particular problem and not be concerned with the communication, therefore connectors are naturally evolved from the needs of components programmers. Because there exist many communication middlewares that are typically incompatible between each other and even most of them are not so simple for use, the software connectors have grown up from demands of some unification and simplification of inter-component communication. Thus natural requirements on software connectors is that they have to be highly configurable, easy to use and providing good encapsulation of the underlying communication.

Connectors usually span over more address spaces because of their inherently distributed nature. Figure 3.1 shows an application which consists of four components spread over three address spaces. Each connector element realizing the communication between the server and the client is generally divided into two connector units (the server one and the client one).

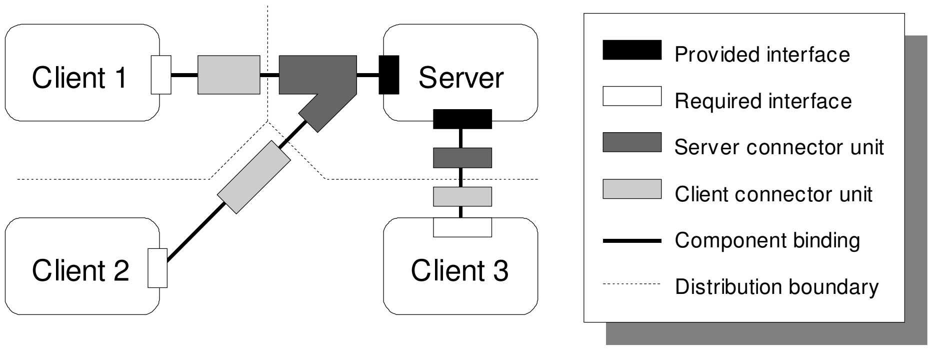


Figure 3.1: A sample component application using connectors

As a side effect connectors besides pure communication can be adding some extra features to a connection (monitoring, logging, security etc.).

Because manual writing of connectors would be surely hard and inflexible work there exist generators which are capable of creating software connectors on the basis of some high level prescription. The main idea can be compared to the code generation from the Interface Description Language (IDL) where a language-independent definition of interface is generated into some concrete programming language.

In this work we are aiming on the connector generator [29] being developed on Charles University in Prague. This generator is highly configurable and allows code to be individually adjusted and optimized specially for each component. Primarily it produces Java classes but its architecture is modular and extensible enough to support also other languages.

3.1 Modelling connectors

Connectors are modeled using small pieces that can be nested. Thus they form tree structures. Leaves are the most specific and doing one concrete thing. They are grouped together into nodes which represent more and more abstract functionality. The toplevel nodes forms big complex units that can be connected together.

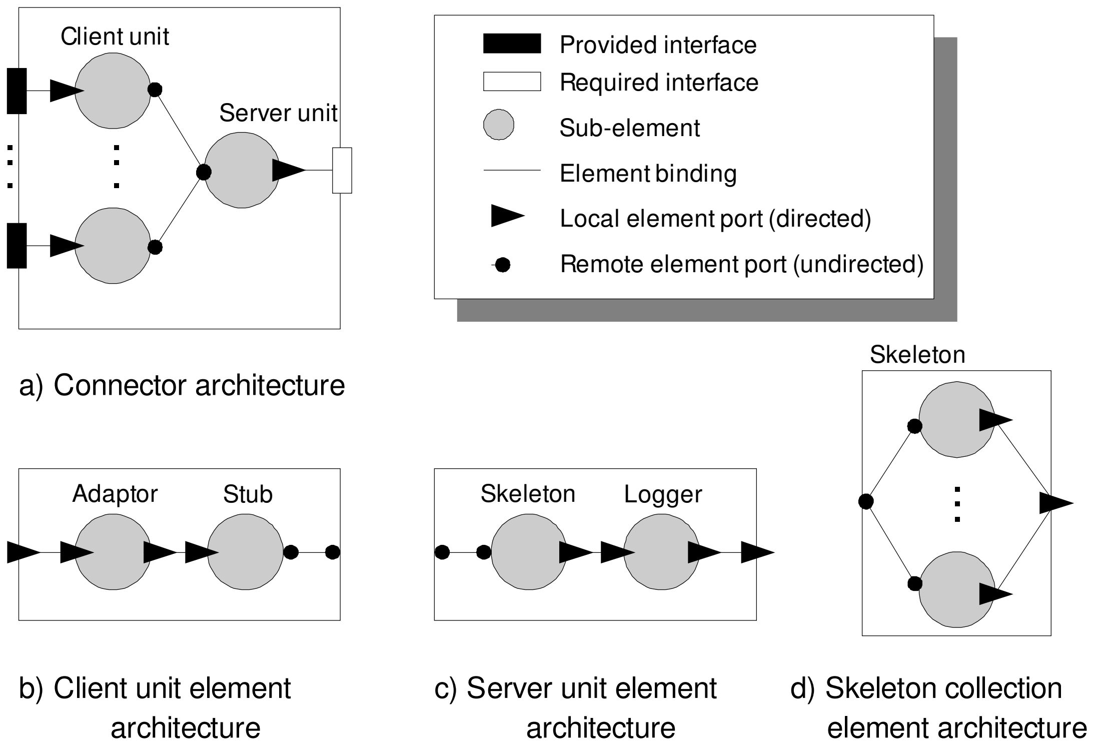

Client unit
Server unit
Provided interface
Required interface
Sub-element
Element binding
Local element port (directed)
Remote element port (undirected)
a) Connector architecture
Adaptor
Stub
Skeleton
Logger
Skeleton
b) Client unit element architecture
c) Server unit element architecture
d) Skeleton collection element architecture

Figure 3.2: A sample connector architecture

A *connector architecture* (see Figure 3.2a) defines the first level of nesting. It says how its main parts (toplevel nodes) are joined together. These main parts are so-called *connector units*. The connector unit typically communicates on one side locally with a component attached to it and on the other side remotely with a corresponding connector unit.

The basic building entity of a connector is a *connector element*. It is shown in Figure 3.2b, 3.2c and 3.2d. Is is a box that can contain some sub-elements. Connector units (toplevel nodes) are by itself a connector element and they can be next subdivided. It can be represented as a tree.

Each connector element has some *connector type*. The connector type designates which *ports* the corresponding connector element will have. Ports are access points through which the elements can be connected between each other. The connector type can be viewed as a black-box which have strictly prescribed its interfaces but what is inside is now visible. In our model we distinguish three kinds of ports:

- **provided ports** – define points accessible for other elements (in Java they can be realized as interfaces which the particular class provides – methods can be called through it).
- **required ports** – each required port must be connected with some appropriate provided ports of some other element (again in Java it is something like the situation where some references in a class have to be initialized in order the class could be used – otherwise null reference exception would probably be thrown).
- **remote ports** – provided as well as required ports are intended to be used within one address space (in one JVM). In contrast remote ports provide interconnection between separate JVMs.

There are also two types of bindings:

- **local binding** - each required port has to have corresponding provided port. Such a binding is taken as directed, the caller with required port calls some method on a provided port, so the direction is always from “required” to “provided”. The call is proceeded in one address space.
- **remote binding** – represents a complex communication typically realized by some middleware. In out model we view this binding as undirected and as a “hyper-edge” – it allows to connect several ports together.

Connector elements can be either *primitive* or *composite*. Primitive elements are basic entities, they are no more subdivided and contain some piece of implementation. Composite elements group elements together, they prescribe element types (not concrete elements) of its sub-elements and bindings between them and thus they serve as a container. Sub-elements of a composite element can be both primitive or composite. It forms a tree structure, the level of nesting is potentially unlimited, but there should not be circular references.

Picture 3.2a shows the topmost architecture of the connector. It says that the connector has one server unit and any number of client units. Both server and client units are ONLY connector types, it does not force any concrete implementation, it defines that there must be included some element of the appropriate type. Picture 3.2b then shows an internal structure of one concrete client unit. There can be more architectures of client units and then the most appropriate will be chosen by the architecture resolver (see later).

There are some restrictions on where provided, required and remote ports can be. Connector units (elements on the first level) can contain only remote ports even if two connector units reside in one address space. In contrast inside a unit there may be only local bindings between sub-elements (provided/required ports). If a sub-element has a remote port it has to be delegated to its parent and so on until the enclosing connector unit is reached.

Besides that each port is of some particular type (provided, required or remote), they must also have assigned a *signature*. The signature is a string with functional form: `operator(operand1, operand2, ...)`. The `operator` is in lowercase and can represent some special function, e.g. `rmi()` which adapt ordinary Java interface for remoting, or just have only semantic meaning without physically modifying the operands. The operands can be either nested operators, strings surrounded by quotes or variables (in uppercase).

Only ports with signature that can be *unified* can be joined together. This simple mechanism avoids linking of two incompatible ports.

The *unification* is very similar to the unification from the Prolog [30] programming language. A signature of the one side of binding is unified with the signature from the other side of that binding.

Variable names are local to a connector element. That means for example if a variable `I` occurs a number of time in the port signatures of one connector element the are “bound” together. But the same name `I` in some other element has no association, they do not interfere.

In Figure 3.3 is depicted RMI stub with its two ports – the provided `call` and the remote `line`. The `call` port has signature `java_iface(I)` where the letter `I` denotes formal parameter. With this particular function (`java_iface`) it mu be a name of some existing java interface, e.g. string `'BussinesIFace'`. This formal parameter is bound with formal parameter of `line` port because they have the same name.

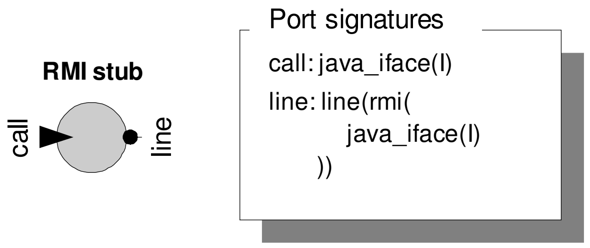


Figure 3.3: Formal signatures of ports

On the other side of the remote binding (of the `line` port) is RMI skeleton's `line` port with the same signature.

Thus the interface names can be propagated through the connector via the unification. It is illustrated in Figure 3.4 where all formal parameters have been already substituted with the '`BussinesIFace'` interface.

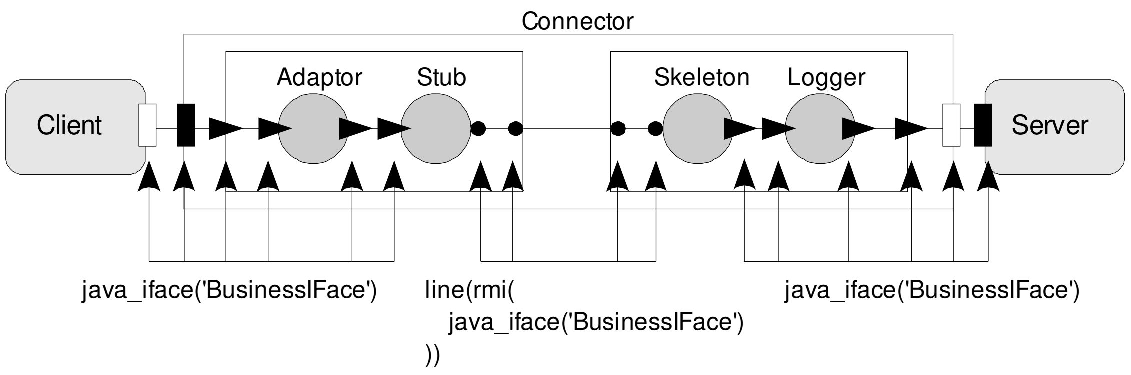

Connector
Adaptor
Stub
Skeleton
Logger
Client
Server
java_iface('BusinessIFace')
line(rmi(
java_iface('BusinessIFace')
))
java_iface('BusinessIFace')

Figure 3.4: An example of interface propagation through the connector

3.2 Communication styles

A *communication style* is basically a way how particular *connector units* (see Chapter 3.1) interact with each other, their kind of communication. It naturally stems from families of protocols or technologies that have some common patterns. For example Java RMI or XML-RPC [31] both belong to the *method invocation* communication style because their common feature is calling methods as a kind of interaction, or TCP/IP and RTSP (Real Time Streaming Protocol) can be considered as a *streaming* communication style.

Naturally different communication styles require different *connector* architectures (see Chapter 3.1) because of their various needs and approaches to the communication. In this thesis we elaborate two styles because Java RMI and CORBA belong to the *method invocation* and JMS represents *messaging*.

3.2.1 Method invocation

By this communication style is meant the method calling on objects. Methods can be invoked on a local object as well as on a remote one. Mostly a user do not have to even know about it, the behavior is the same – a client code holds a reference to some object (local or remote) and calls a method through it.

Because of a flexibility the client do not have to hold direct reference to the server object, but instead a proxy that is redirecting the call. Between the client and the server component can be any number of elements. They are in general divided into two groups – one which reside on the client side and the others on the server side.

In figure 3.5 is depicted an architecture used within the method communication style.

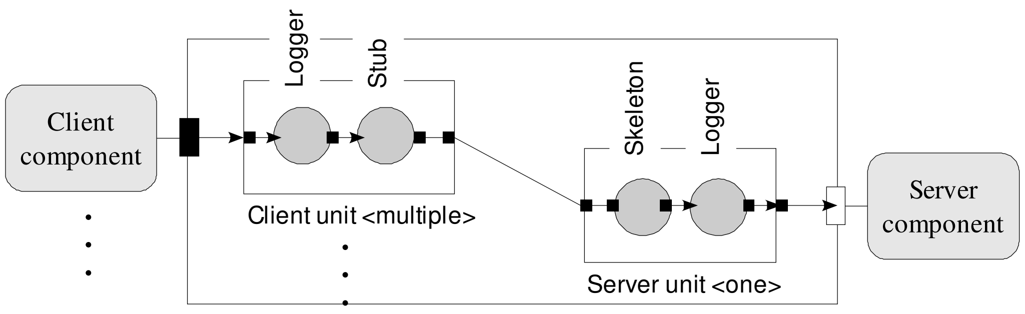


Figure 3.5: The method invocation architecture

As it can be seen a client unit can have multiple instances and the server only one. It is a typical relation n..1. The client unit is connected on one side with a client component through a local binding and on the other side remotely with the server. In the picture is only a simple architecture where the client unit have only two sub-elements – the *logger* logs thorough calls and the *stub* provides a connection with the server.

Into the client unit as well as into the server unit there can be added any number of sub-elements. Figure 3.6 shows an illustrative more robust architecture of the server unit where the synchronizer takes control of threading policy, transaction adapter manages transactions, server adapter alters minor incompatibilities between client and server interfaces (the *Decorator* design pattern) and server interceptor monitors a thorough communication.

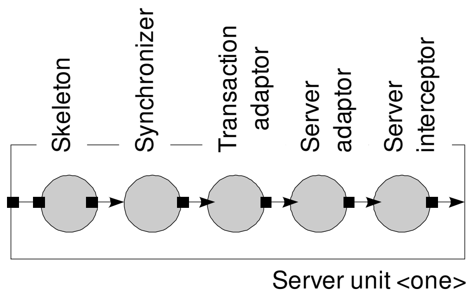


Figure 3.6: The robust server unit in the method invocation

3.2.2 Messaging

Messaging also uses client/server architecture. Here the clients can produce messages (producers) and receive them (consumers). There are different architectures for the two messaging types (point-to-point vs. publish/subscribe). In Figure 3.7 is depicted the connector architecture for the publish/subscribe messaging. It shows a client component which can figure in both roles – as a producer as well as a consumer. It depends which client's unit port the component uses. It can also use both of them and act as a producer and consumer at the same time.

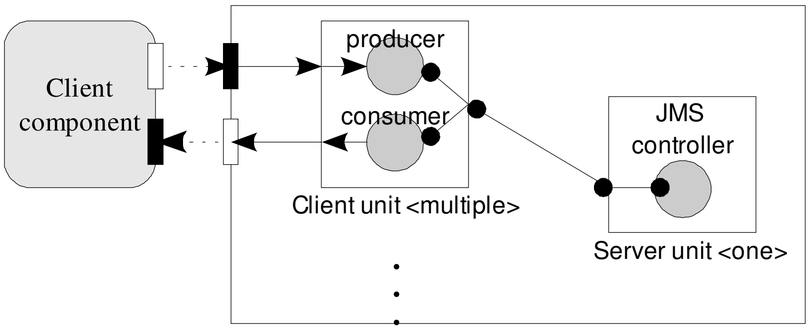


Figure 3.7: An architecture of the messaging communication style

When some producer creates and sends a message, the server forwards it to the all registered consumers. Consumers are passive, they are not polling the server, instead they are automatically given the message, therefore the consumers must provide some interface through which the client unit can pass it (e.g. through some function like `onMessage(Message m)`).

But that is not the case with the point-to-point messaging. Clients have to explicitly ask for a message. When no message is actually in the server's queue, the client is blocked until the message arrives or optionally specified timeout expires.

3.3 Generating connectors

The generation process is divided into two steps. At the beginning a user has to provide a high level configuration of the connector. This configuration is designed to be human readable and easily editable. Besides the list of all components in the application ant their interfaces the high level configuration contains also the communication style which should be used (method invocation, messaging, ...) and a set of non-functional properties (NFPs) (in fact the communication style is set also through NFP).

The NFPs is a set of named attributes. It describes the configuration in a declarative way. The attributes are written in a dot notation and can be used in logical predicates like the following example shows:

(monitoring.level = 'brief' && monitoring.filename = 'app.log')

An *architectural resolver* must build the resulting architecture (which connector elements will be used and how they will be connected between each other). It reads a high level configuration file provided by the designer of an application, *connector and element architectures* (see Chapter 3.1) and capabilities of the particular hosts where the components will run. Based on that the architecture resolver is supposed to find “the best” configuration that satisfies all the requirements.

The resolution is done based on the Prolog programming language because of its logic nature and capabilities. It has turned out that it is really suitable for this kind of task.

Figure 3.8 shows the whole generation process:

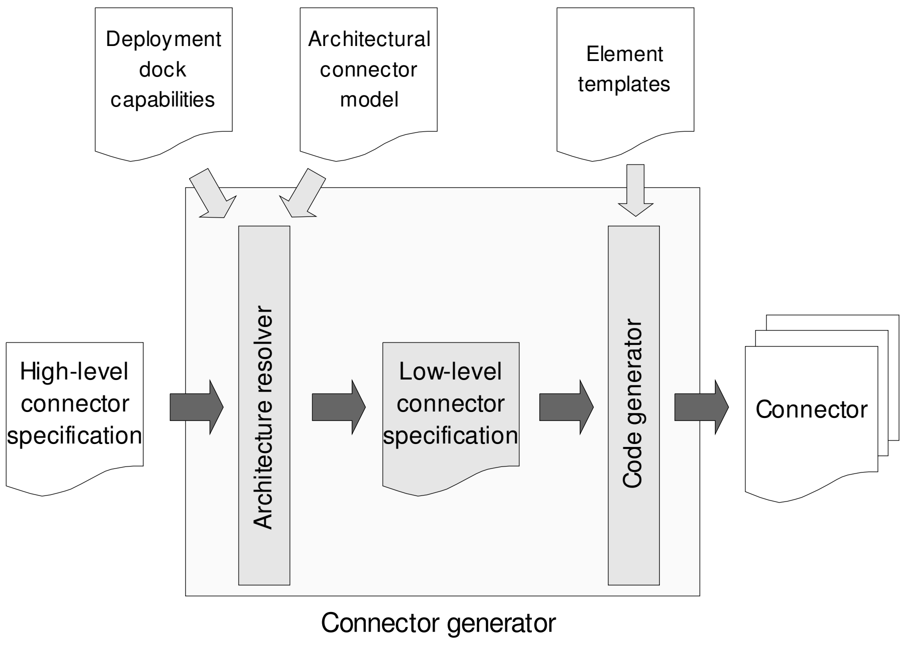

Deployment dock capabilities
Architectural connector model
Element templates
High-level connector specification
Architecture resolver
Low-level connector specification
Code generator
Connector
Connector generator

Figure 3.8: The generation process

The *low-level connector specification* produced by the architecture resolver has already have assigned all concrete connector elements that will be needed.

The *code generator* reads that configuration and also the *element templates* that contain parts of source code that must be adjusted depending on the resolved architecture. At the end are all generated sources compiled into an executable code and created libraries which forms a connector.

Chapter 4

Goals elaborated

The connector generator in only a framework that helps with the process of the component development. It has facilities for engineering a connector's architecture, deployment and not least for distribution. But it is meaningful only with a connection with some existing middleware which provides the real communication in the generated connectors.

The goal of the thesis is to integrate various middlewares in the connector generator. We have chosen three technologies of middlewares because of reasons given in Chapter 2. These concrete technologies are Java RMI, CORBA and JMS.

There was a partial support for Java RMI, but the others were not implemented at all. The aim of the thesis is to extend the existing connector generator and provide support for generating and deploying RMI and CORBA-based connectors and also connectors using the JMS API.

Thus the main goals of this thesis are:

- complete the support for Java RMI
- propose and implement support for CORBA so that it would be able to do similar things as Java RMI
- choose a concrete JMS implementation and integrate it with the connector generator

4.1 Complete the full support for Java RMI

At the time of writing this thesis there was only a partial support for Java RMI. There was a good support for primitive data types as well as there had been solved passing of remote references. Support for object types was not any so we want to propose a solution which makes the passing of object types within Java RMI possible.

4.2 Implement support for CORBA

There was no support for CORBA so we attempt to implement it in that way it will be able to do similar things as Java RMI (primitive and complex data type passing as well as remote reference passing).

CORBA can be integrated in the connector generator in number of ways. A natural approach to CORBA development is to begin with some IDL (Interface Description Language, see Chapter 2.2) and from that create an interface in some concrete language.

But since the connector generator has a good support for the Java language it could be more suitable to begin with Java interfaces. We want to discuss this issue and propose and realize some implementation.

4.3 Integrate the JMS API

The JMS was intentionally chosen to implement different communication style. Java RMI as well as CORBA belong to method invocation communication style in contrast to JMS representing messaging.

Integrating JMS API and with it the messaging communication style seems that it can be a good example of the strength and flexibility of the generator. It is a good opportunity to demonstrate how to extend support by a new architecture.

It will require to create number of configuration files, add new templates (see later) and generators and maybe the core of the connector generator will have to be slightly adjusted to meet JMS demands.

Chapter 5

Connector generator overview

The connector generator within its design is very good extensible and modifiable. All configuration files are in XML (eXtensible Markup Language) [32] which is human readable and a good structured format. The main configuration files reside under `org/objectweb/dsrg/congen/conrep` folder and its subfolders and the one of the most important is the `conf.xml` file shown in Figure 5.1. It contains list of registered connector architectures, element types and elements itself which are all described in the following subsections.

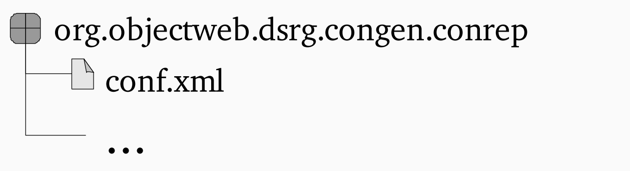


Figure 5.1: Location of the main configuration file

5.1 Connector architectures

As discussed before *connector architectures* (see chapter 3.1) defines the first level of nesting, how particular *connector units* are joined together and their cardinality. In the connector generator only two cardinalities are permitted: *one* and *multiple*.

The cardinality '*one*' is usually used within some server entity which is typically single or there is a limited number of them in contrast to client entities which are in general more, so the cardinality '*multiple*' comes in handy.

Figure 5.2 shows the location of the configuration files related to connector architectures. When there is a need to add support for another communication style there must exist a corresponding configuration file under that folder. But that is not,

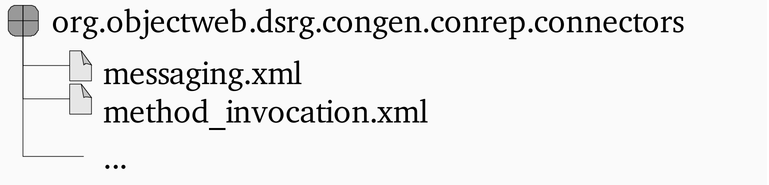


Figure 5.2: The configuration files for connector architectures

enough it also has to be registered in the main configuration file `org/objectweb/dsrg/congen/conrep/`conf.xml as demonstrates Figure 5.3. It says that the connector generator will be looking for the files `method_invocation.xml` and `messaging.xml`. As their names prompt `method_invocation.xml` contains connector architecture for the *method invocation* communication style (see Chapter 3.2.1) and `messaging.xml` for the *messaging* communication style (see Chapter 3.2.2).

```
<configuration>
       ...
       <connector file="method_invocation.xml"/>
       <connector file="messaging.xml"/>
       ...
</configuration>
```

Figure 5.3: org/objectweb/dsrg/congen/conrep/conf.xml

The structure of the configuration files is quite simple, here is an example of a definition of the method invocation communication style:

```
<connector name="method_invocation">

  <architecture>
    <unit name="client_unit" type="rpc_client_unit" cardinality="multiple"/>
    <unit name="server_unit" type="rpc_server_unit" cardinality="one"/>

    <binding>
      <port element="client_unit" port="line"/>
      <port element="server_unit" port="line"/>
    </binding>
  </architecture>

</connector>
```

Figure 5.4: org/objectweb/dsrg/congen/conrep/connectors/method_invocation.xml

The root element `<connector>` has one attribute which defines the name of the architecture. The user can then specify in the high level configuration files which architecture should be applied in the generation process.

The element `<architecture>` defines the layout of the connector architecture. It prescribes the first level of nesting – the elements `<unit>` represent the root nodes of the tree hierarchy and `<binding>` describes their mutual relationship. Each `<unit>` has a name which is referenced in `<binding>`, a type and a cardinality.

The type do not force any concrete implementation of the element, it is rather a class of elements. It says that in the place of `<unit>` can be any element of that type. Which concrete implementation it should be is decided by the architectural resolver when generating particular connector (see Chapter 3.3).

The `<binding>` element describes how the units are connected together, through which ports. In Figure 5.4 there are only two ports, but <binding> can join also a bigger number of them, it is like a hyper edge. Also note that there is no direction nor any signatures. Directions are not stated because between units on the first level of nesting there are only remote bindings which do not have any. And signatures are inherited from subelements, there is no need to define them here because the architecture is as abstract as possible. There is a certain freedom in that how the resulting connector will look like. The architecture in only a skeleton into which the concrete elements can be inserted.

5.2 Element types

Element types (e.g. `rpm_client_unit`) can be viewed as a black-box which have only defined ports (provided, required and remote) and it is not known what is inside. It can be imagined as an abstract class or interface from programming languages. It says how it should look like but it does not force any concrete realization. In places where some type is expected can be any element of that type.

Besides a configuration XML file describing the element type it must be also registered in the `conf.xml` as in Figure 28.

```
<configuration>
       ...
       <element-type file="rpc_client_unit.xml"/>
       <element-type file="rpc_server_unit.xml"/>
       ...
</configuration>
```

Figure 5.5: Registration of element types

The structure of the configuration files is simple. They contain only enumeration of port names a their type (either `required`, `provided` or `remote`):

```
<element-type name="rpc_client_unit">

  <port name="call" type="provided"/>
  <port name="line" type="remote"/>

</element-type>
```

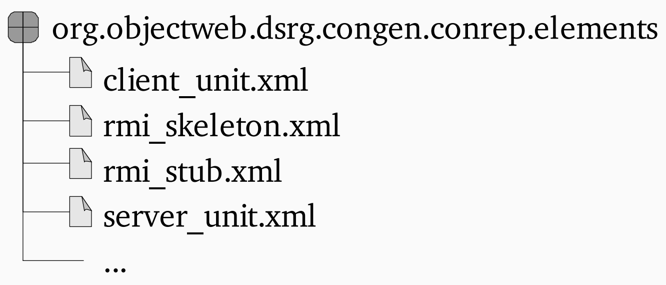
Figure 5.6: The configuration file of the `rpc_client_unit` element type

Figure 5.6 shows the type `rpm_client_unit` that has two ports: the provided `call` which serves as local communication channel with an attached client component and the remote `line` which is used in remote binding (see `<binding>` element in Figure 5.4).

The name of the element type (specified through the attribute `name`) is referenced in the connector architectures (see previous Chapter 5.1, especially the attribute `type` of the element `<unit>`).

The port names are referenced in bindings, namely in connector architectures to define relations between connector units and in the elements (see next Chapter 5.3) to specify connections between their subelements.

5.3 Elements



```
org.objectweb.dsrg.congen.conrep.elements
├── client_unit.xml
├── rmi_skeleton.xml
├── rmi_stub.xml
├── server_unit.xml
└── ...
```

Figure 5.7: A location of configuration files for elements

When element types were compared to interfaces from high level programming languages, the elements can be compared to implementations of that interfaces. An element is like a gray-box, it is more specific than an element type. Besides that it has assigned an element type (defining its ports which are “entry points” to the element) it also defines the internal structure of the unit, its non-functional properties (for NFPs see Chapter 3.3) and scripts related to the code generation.

Elements are divided into composite and primitive elements, both discussed in the following subsections. Figure 5.3 shows their common structure:

```
<element name="client_unit" type="rpc_client_unit" impl-class="ClientUnit">

  <architecture cost="0">
    ...
  </architecture>

  <nfp-declarations>
         <nfp-mapping name="communication_style" value="method_invocation"/>
  </nfp-declarations>

  <script>
    <command action="jimpl">
        <param name="generator"
            value="org.objectweb.dsrg.congen.elemgen.
                generators.CompositeGenerator"/>

        <param name="class" value="ClientUnit"/>
        <param name="template" value="compound_default.template" />
    </command>

    ...

  </script>

</element>
```

Figure 5.8: Structure of element configuration file

The root tag `<element>` has these parameters:

- **name** – name of the element.
- **type** – the type denotes which outward ports the element will have, they serve

as an entry points. Any other ports inside the element are not visible. For element types see Chapter 5.2.

- **impl-class** – name of a Java class which will contain implementation for this element (after the generation proceeds).

The `<element>` tag also has some subelements:

- **<architecture>** - describes the layout of the element, the bindings between subelements and ports. It slightly differ between composite and primitive elements.

  This tag has also the attribute '`cost`' which expresses approximate consumption of system resources. Higher value means that this element consumes either more memory, processor speed, network bandwidth etc. It is of course very estimative value which do not event have to reflex the reality.
- **`<nfp-declarations>`** - contains list of attributes and their assigned values. It describes some properties of the element, it is a declarative way how to specify required properties of the element. In Figure 5.8 `<nfp-declarations>` says that the element is usable only in *method invocation* communication style.
- **`<script>`** - this section prescribes how the element should be physically generated and compiled. It can utilize development tools like `javac` compiler.

5.3.1 Composite elements

Composite elements contain some subelements. The <architecture> tag then describes how are these subelements joined together. Here is an example of an architecture taken from the `logged_client_unit.xml`:

```
<architecture cost="0">
  <inst name="logger" type="logger"/>
  <inst name="stub" type="stub"/>

  <!--
    Element bindings:

    <this>.call (local, provided)  -delegation->  <logger>.in (local, provided)
    <logger>.out (local, required) -binding->     <stub>.call (local, provided)
    <stub>.line (remote, required) -subsumption-> <this>.line (remote, required)
  -->
  <binding port1="call" element2="logger" port2="in"/>
  <binding element1="logger" port1="out" element2="stub" port2="call"/>
  <binding element1="stub" port1="line" port2="line"/>
</architecture>
```

Figure 5.9: An example of an architecture of a composite element

The list of the tags `<inst>` says which subelements the element contains. They are named (the name is then referenced from the `<binding>`) and assigned a type which restricts a set of substitutable elements.

Next the `<architecture>` contains also the `<binding>` elements. It describes connections between the subelements. The connection is oriented and it can be imagined as an arrow pointing from `element1` and its port `port1` to `elements2`'s `port2` like in the picture:

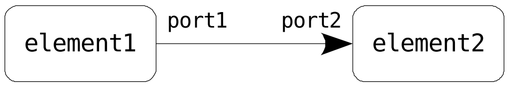


Figure 5.10: Binding between subelements

The `element1`, `port1`, `elements2`, `port2` are all the attributes of the `<binding>`. If `element1` or `element2` are omitted, the parent's port is supposed to be used (specified with corresponding `port1` or `port2`).

The attribute `cost` (from figure 5.9) with zero value may look like a little strange. There is the zero because the total cost is computed as the cost of an element (0 in this case) plus sum of costs of all its subelements. The composite element shown in Figure 5.9 serves only as an container that do nothing except gluing its subelements, thus the resulting cost depend on their overall cost.

Here is a graphical illustration how the elements from Figure 5.9 are binded together:

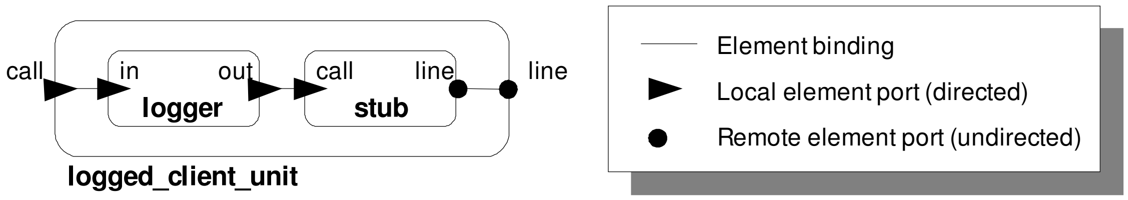


Figure 5.11: A graphical illustration of the composite element from Figure 5.9

5.3.2 Primitive elements

Primitive elements contain no other subelement. They are the main building blocks of the connector. Figure 5.12 shows an example of an primitive element:

```
<architecture cost="5">
    <port name="call" signature="I"/>
    <port name="line">
      <signature-entry ref-name="rmi" type="client" signature="rmi(I)"/>
    </port>
</architecture>
```

Figure 5.12: An example of an architecture of a primitive element

Here `<architecture>` contains only the XML tags `<port>` which to each element's port assign some signature. It is in contrast to composite elements (see previous Chapter 5.3.1) where the signatures are not specified but rather they are inherited from their subelements.

But there is a slightly different specification of signatures between the local and remote ports.

The local ports contain only textual value within `signature` attribute, no other informations are needed.

But the remote ports are more complex. Each remote port is rather a collection than one single entity. The collection of signatures (used for remote bindings) is formed by `<signature-entry>` elements.

The signature of one entry is captured by the `signature` attribute and its meaning is similar to local port signatures.

Next they are differentiated by the attribute `type` which can have three values – `client`, `server` and `both`. It denotes how the certain entry is participating in the binding, whether it is providing some middleware reference (server), or just using it (client) or both cases.

And finally the attribute ref-name is used to distinguish individual references in the collection.

5.4 Binding elements

In order that elements can communicate between each other they must be somehow linked together. Elements residing in one address space are connected only via “ordinary” references (like pointers from low level programming languages), they are not described here, this chapter is aimed on gluing elements from distinct JVMs.

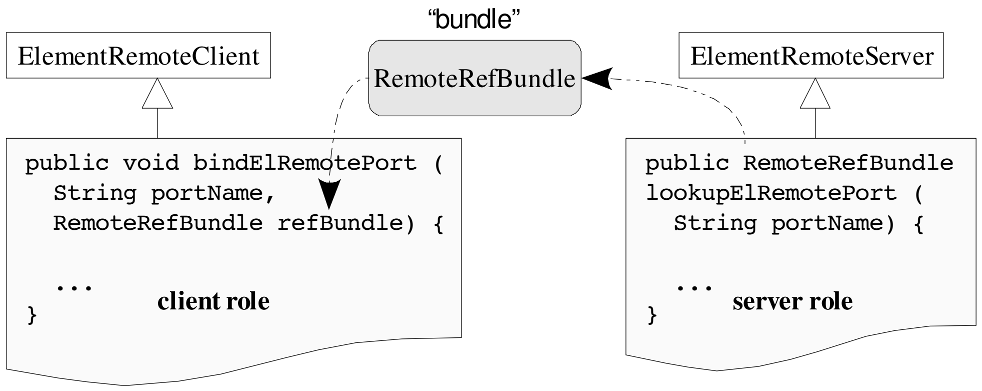


Figure 5.13: Linking two elements residing in separate address spaces

Remote references are mediated via the special object of type `RemoteRefBundle` (see Figure 5.13). It contains list of named values (String `key : String value).` In order that the process of linking in the connector could be automatic, various elements must implement particular interfaces. When linking two elements together one is in a client role and the other acts as a server:

- Element in the server role is providing some service. It have to expose a remote reference to itself so that clients can get use it. It must implement the `ElementRemoteServer` interface which contains the method `lookupRemotePort().` It is supposed to return a remote reference (associated with some key) that is inserted into the global “bundle”.

- Clients have to implement the `ElementRemoteClient` interface containing the method `bindElRemotePort()`. That method is given the global “bundle” as a parameter and the client should extract (using the appropriate key) and retain the remote reference.

5.5 Element generators

The *architecture resolver* (see Figure 3.8) produces a prescription for the low level *element generator* that creates some real source code. These sources are then compiled into libraries which forms the connector.

Each *connector element* can have associated specific generator that will be used in the generation process. It is configured in the element configuration file (see Figure 5.8) inside the `<command action="jimpl">` section which contains parameters specific for compilation:

```
<command action="jimpl">
        <param name="generator"
            value="org.objectweb.dsrg.congen.elemgen.
                generators.CompositeGenerator"/>

        ...
</command>
```

Figure 5.14: The build commands used for one specific element. Each element is configured separately.

So there can be different generators e.g. for `local_stub` and for `rmi_stub`. These generators form an hierarchy, on the top is `BaseGenerator` and all the others inherit from it. There are also two general generators for primitive elements (the `PrimitiveGenerator` Java class) and for composite elements (the `CompositeGenerator` class). These classes are supposed to be further subclassed to achieve a more specific behavior suitable for concrete connector element.

Figure 5.15 shows the generator's hierarchy. Core generators are for common use, they are not specialized, but provide basic functionality. The figure also shows element specific generators which extend `PrimitiveGenerator`.

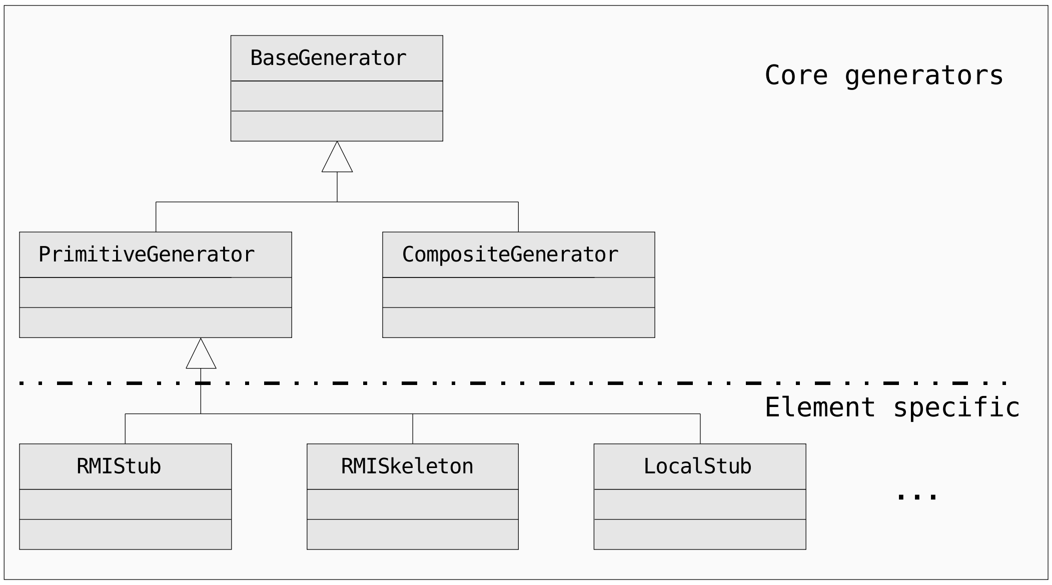


Figure 5.15: A hierarchy of generators

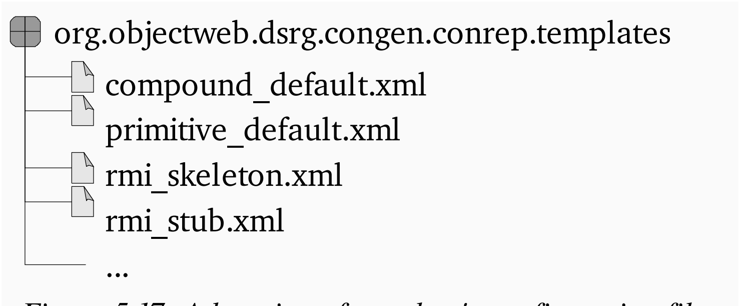
`BaseGenerator` contains the function `getContent()` which is then inherited by all other generators. It takes a string as an input and returns also string, its purpose is to substitute one string for another which is heavy used within templates (see next Chapter 5.6). The other generators can override this function to add more substitutable words.

Figure 5.16 shows the structure of the function `getContent()`. Its body is a system of if clauses which works similar to the `switch` statement. But the `switch` statement does not work with a string argument therefore it is solved in this way:

```
public String getContent (String identifier)
       throws TemplateProcessingException {

       if ("PACKAGE".equals (identifier)) {
              return destPkg;
       } else if ("CLASS".equals (identifier)) {
              return destClass;
       } else if ("ELEMENT_METHODS".equals (identifier)) {
              return implementGetElDescription ();
       }

       throw new TemplateProcessingException (
              "Unknown content identifier: "+ identifier);
}
```

Figure 5.16: An overview of the `getContent()` function

5.6 Templates



```
org.objectweb.dsrg.congen.conrep.templates
    compound_default.xml
    primitive_default.xml
    rmi_skeleton.xml
    rmi_stub.xml
    ...
```

Figure 5.17: A location of template's configuration files

A template is hybrid source code. It contains some static parts which are always the same and dynamic parts that are generated as necessary for the concrete connector.

It is a normal textual file but some words have special meaning, they can be compared to macros from some programming languages. They are escaped at the beginning and at the end by the letter '%'. These special words are then substituted by the code generator (see Chapter 3.3) using the function `getContent()` (see Chapter 5.5) and the template is adapted by the needs of the concrete connector element.

Figure 5.18 shows how such a transformation can look like in practice:

```
package %PACKAGE%;

public final class %CLASS%
       extends
       javax.rmi.PortableRemoteObject
       implements
       org.objectweb.dsrg.connector.ElementRemoteServer,
       org.objectweb.dsrg.connector.ElementLocalClient,
       %IMPLEMENTS% {

       protected %TARGET_INTERFACE% target;
       ...
}
```

Processed by the code generator

```
package generated.A00000002;

public final class CorbaSkeleton
       extends
       javax.rmi.PortableRemoteObject
       implements
       org.objectweb.dsrg.connector.ElementRemoteServer,
       org.objectweb.dsrg.connector.ElementLocalClient,
       generated.A00000003.Interface {

       protected congentest.Compute target;
       ...
}
```

Figure 5.18: An example of a code generation from some template

It is seen that the macros `%PACKAGE%`, `%CLASS%`, `%IMPLEMENTS%` and `%TARGET_INTERFACE%` have been substituted by the concrete Java entities.

The exact list of defined macros depends on a concrete generator used for some connector element. When an undefined macro is used the generator throws an exception. The Figure 5.18 shows only a simple case when the macros are substituted by one Java identifier. But macros can denote also more complex Java entities, even the whole Java classes.

5.7 Type system

In a heterogeneous environment the connector generator has to deal with various types. Each language and middleware is using its own special data types that are incompatible with the others. But because connectors are supposed to join different components by utilizing miscellaneous middlewares, the connector generator must provide some model for working with different types.

The connector generator must have some concept how to hold information about distinct types. Internally it defines three general interfaces - `InderfaceDef`, `PrimitiveDef` and `ArrayDef`. How the names suggest it is for representing interfaces, primitives and arrays. Each specific type system then subclasses these general interfaces. Figure 5.19 shows classes for representing Java types:

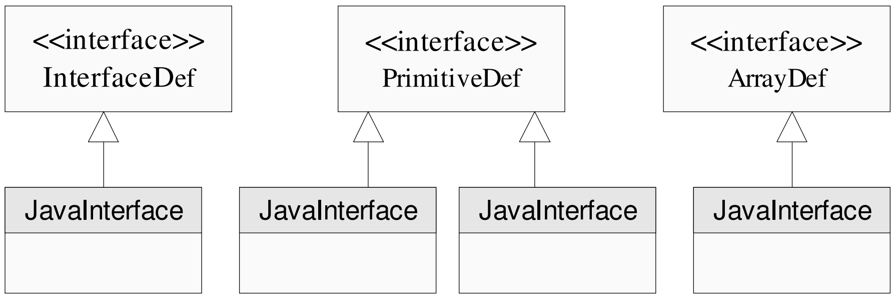


Figure 5.19: An extension of the abstract type model

The types are also used in signatures of the ports. For example type operator `java_interface('BusinessInterface')` that can be used in signature loads existing Java interface (through the Java reflection). Then there can be some operators for modifying it, e.g. `rmi(java_interface('BusinessInterface'))` adjust the operand (it must be a Java interface) in a way that it can be used as a remote interface in RMI.

The type system can also define operators for converting types between different languages, for example `java_to_idl()` or `idl_to_java()` for transforming interface between Java and IDL.

The system is designed in order to be very good extensible. Thus when some new operator is needed it can be easy implemented and integrated into the existing type system.

The connector generator has also some special internal functions that operates upon some specific types. For example `JavaInterfaceWriter.write(Type type, String destPackage)` generates a Java source for a Java interface and writes it into a file. The parameter `Type` must contain description of Java interface that is supposed to be written, if different type is passed runtime exception will be thrown.

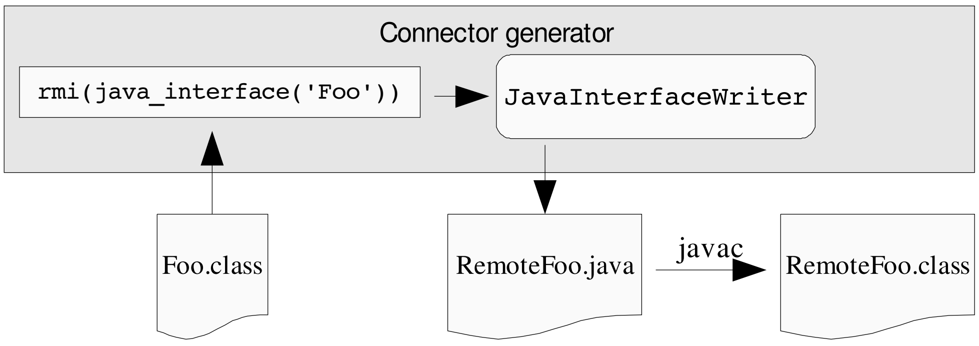


Figure 5.20: A demonstration of the usage of the type operators and `JavaInterfaceWriter`

Figure 5.20 shows how the type operators can be used. At the beginning there is a Java type `Foo`. The connector generator wants to use it in RMI but it does not satisfy all the requirements (see Chapter 6.1) for usage in the remoting and thus it must be modified. It is loaded into the generator through the `java_interface()` operator and adjusted using the `rmi()`. The resulting java interface is passed to the `JavaInterfaceWriter` which writes it onto the filesystem and finally the connector generator invokes a Java compiler for creating an executable code.

Chapter 6

Implementing support for Java RMI

6.1 Java RMI Overview

Java RMI is a technology for distributed computing. RMI applications mostly consist of two separate program – a server which is providing some service and a client which is using it. The server has to make some of its object available in order clients can invoke methods on them. Such objects are referred as *remote objects*.

When designing an application architecture a programmer should first define remote interfaces through which some "remote services" will be available. A remote interface in terms of Java RMI is a Java interface which satisfies some special requirements:

- A remote interface must extend from the interface `java.rmi.Remote`.
- Each method of the interface must contain `java.rmi.RemoteException` in its throw clause (it applies also for all inherited methods).

Here is an example of a valid remote interface:

```
public interface Test extends java.rmi.Remote {
        void doSomething() throws java.rmi.RemoteException;
}
```

Then the programmer can implement some remote objects. A remote object must implement at least one remote interface in order clients can reference it. From the client's perspective the remote objects are accessible only through the remote interfaces, they never hold directly the implementation classes. Instead they have a reference to a *stub* which acts as a proxy to the remote object. The stub implements the same interfaces as the remote object which it represents. All method invocations on the stub are redirected to the server.

When the server's remote objects are implemented clients can use them. With earlier versions of Java (prior to Java Platform, Standard Edition 5.0) the `rmic` tool was additionally needed when developing clients to generate stubs. Since Java supports a dynamic generation of stub classes at runtime, this step is no longer needed.

In order that clients can locate remote objects there exists the tool `rmiregistry`.

When the server wants to make some remote object accessible it calls the registry to bind a name with the object. Clients can then query the registry for that name to obtain a reference to the associated remote object. The other way how a client can obtain a remote reference is through an other remote invocation where a method can return it.

Because clients can pass to the remote method also objects it may happen that the server will not have a class definition for that objects. In order the server can operate with them it must obtain their class files. That is the other important feature of Java RMI. Clients as well as the server are able to dynamically load the code of classes through the web server which makes RMI applications more robust and flexible.

But there is a limitation within passing objects across the network. Objects that are supposed to be used in RMI must be *serializable*, that is they must implement `java.io.Serializable` interface.

6.2 Former Support for Java RMI

6.2.1 Overview

Java RMI belongs to the *method invocation* communication style (see Chapter 3.2.1). Basically it means that there will be one server and many clients components. Each client will be connected with the server through a RMI connector unit. Here is a connector architecture used within Java RMI:

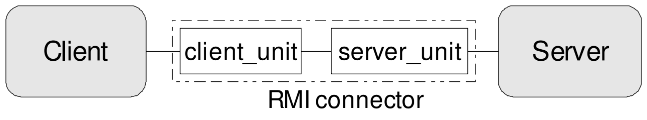


Figure 6.1: A simplified RMI connector architecture

In Figure 6.2 is depicted a more detailed scheme of the RMI connector unit. Each RMI connector unit is divided into the two parts, `client_unit` and `server_unit`,

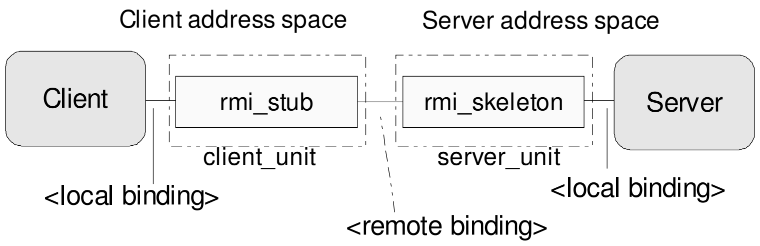


Figure 6.2: Basic division of the RMI connector

which next contain `rmi_stub` (client side) and `rmi_skeleton` (server side). So each client component will be attached through a local binding to `client_unit`, `client_unit` is then connected with the corresponding `server_unit` using remote binding and finally `server_unit` is attached to the server component.

The names `rmi_stub` and `rmi_skeleton` are chosen in that way because the rmi_stub element is generated from the rmi_stub.template file and the skeleton similarly from rmi_skeleton.template (see later).

From an implementation view the client_unit element is just a Java class and the client component must somehow obtain a reference to it. Likewise client_unit have to retrieve the remote reference to server_unit and that must be connected with the server. The user need not even know that in fact all method calls to client_unit are redirected through the network to server_unit and in the end to the server component. The programmer is using the object (client_unit) as if it were a local object.

Figure 6.3 shows a running example of a *business interface* which will be used in the next chapters. It is only demonstrative and very simple, in real application the interface would be probably much more complex.

```
public interface DemoIFace {

    String compute(int arg1, DemoIFace arg2);
    String message();
}
```

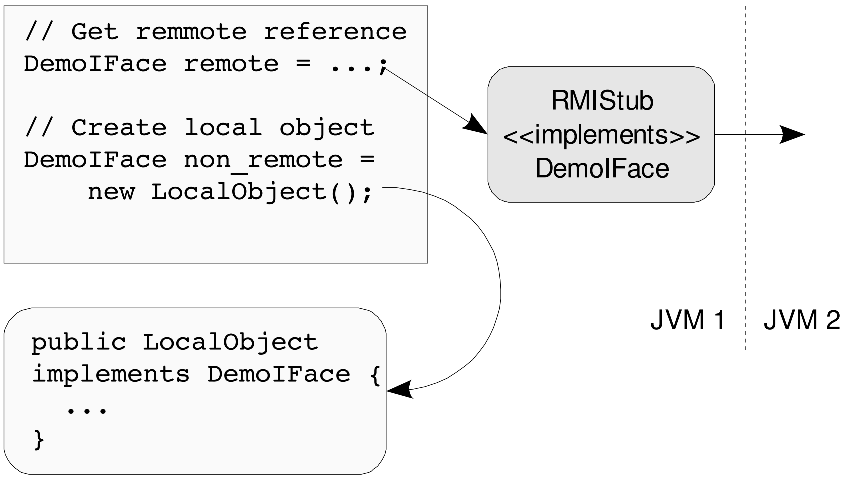
Figure 6.3: A Java interface that will be used in the consequent examples

The term “business interface” is derived from the informal term “business logic” generally used to describe the functional algorithms. It is supposed that application's components implement some business logic and make it available through some interface. We use the the term “business interface” for interfaces which are exposed by the application's components (containing some business logic). It is because we want to distinguish the “normal” interfaces (which are local inside components) from the others which are used for providing remote service. When we speak about DemoIFace as about a business interface we mean that some component is implementing it and providing it as a remote service.

The parameters of the function compute() in Figure 6.3 should demonstrate different kinds of argument passing. The first parameter arg1 of the function is of a primitive type, there is no trouble. The return value String is an object type, but it is so usual (and Java language defines it as serializable) that there is no difficulty with RMI either. But it becomes a little bit tricky when passing a *remote reference* (it will be discussed later).

We use the term *remote reference* in a special meaning. Do not confuse it with RMI remote object. In this thesis we call a reference to a RMI remote object as “RMI remote reference”. Without the word “RMI” it has a connection with the connector. A remote reference has always type of some business interface. When we say for example that a client or object is holding a remote reference of the type DemoIFace we mean that they are holding the appropriate connector's RMIStub (implementing DemoIFace) which is providing communication with the server.

Figure 6.4 shows the difference between a remote reference and an “ordinary” reference.



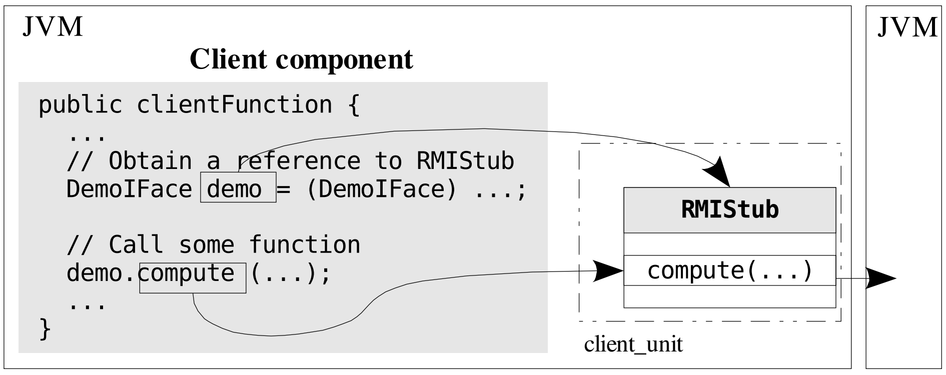
```
// Get remmote reference
DemoIFace remote = ...;

// Create local object
DemoIFace non_remote =
    new LocalObject();
```

```
public LocalObject
implements DemoIFace {
  ...
}
```

Figure 6.4: The difference between a remote reference and an "ordinary" reference

The variable `remote` holds a remote reference because it is linked with the connector which forwards the callings to a remote server. The reference to the object of the `LocalObject` type is not remote because it has no participation in the remote communication. It is just accidentally implementing the `DemoIFace` business interface, but it is a local object.

Figure 6.5 shows an example how the connector can be used from the client component. The client component has to first obtain a remote reference. In fact it will be given the reference to the `RMIStub` which acts as a proxy. All calls through the reference (through `RMIStub`) are forwarded to the server.



```
public clientFunction {
  ...
  // Obtain a reference to RMIStub
  DemoIFace demo = (DemoIFace) ...;

  // Call some function
  demo.compute(...);
  ...
}
```

Figure 6.5: A usage of the RMI connector from the client component

6.2.2 Specifying business interfaces

In order that the RMI connector would be useful with particular application the connector generator which the RMI connector is generated from must be properly configured.

The important thing is to set up which *business interfaces* the application's components provide so that the generator knows for which interfaces the connector should be generated. Is is done through configuring port signatures at the end points of the RMI connector, that is in the place where the connector is joined with the client component (and the server component).

Figure 6.6 shows the situation where the client wants to communicate with the server through the `DemoIFace` business interface. The endpoints are assigned a signature using the type operator `java_interface()` which expects a name of an existing Java interface. The connector generator on the basis of this information generates a specific connector.

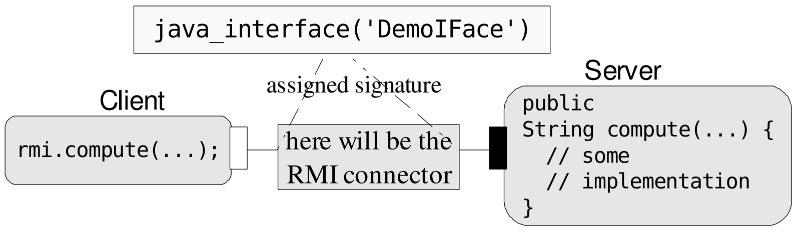
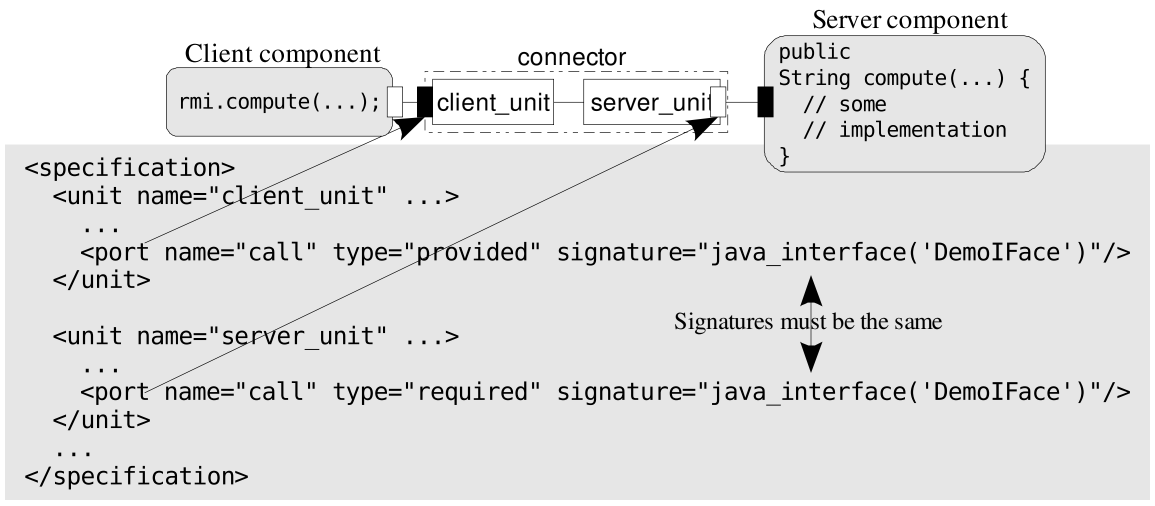


Figure 6.6: Concrete Java interfaces are assigned at the end points

Figure 6.7 shows an example of a possible high level configuration file for the RMI connector. It is a XML file that sets signatures of the end points. A little bit confusing can be that `client_unit` has the `provided` port and `server_unit` has the `required` port. It is because `client_unit` acts as a local server to the client component and `server_unit` is like a client for the server component. If the signatures on the client and the server side do not match the *architectural resolver* (see Chapter 3.3) will be unable to find a proper configuration. If it was necessary to have different Java interfaces on the client and server side there would have to be some adapter that would translate the method callings.

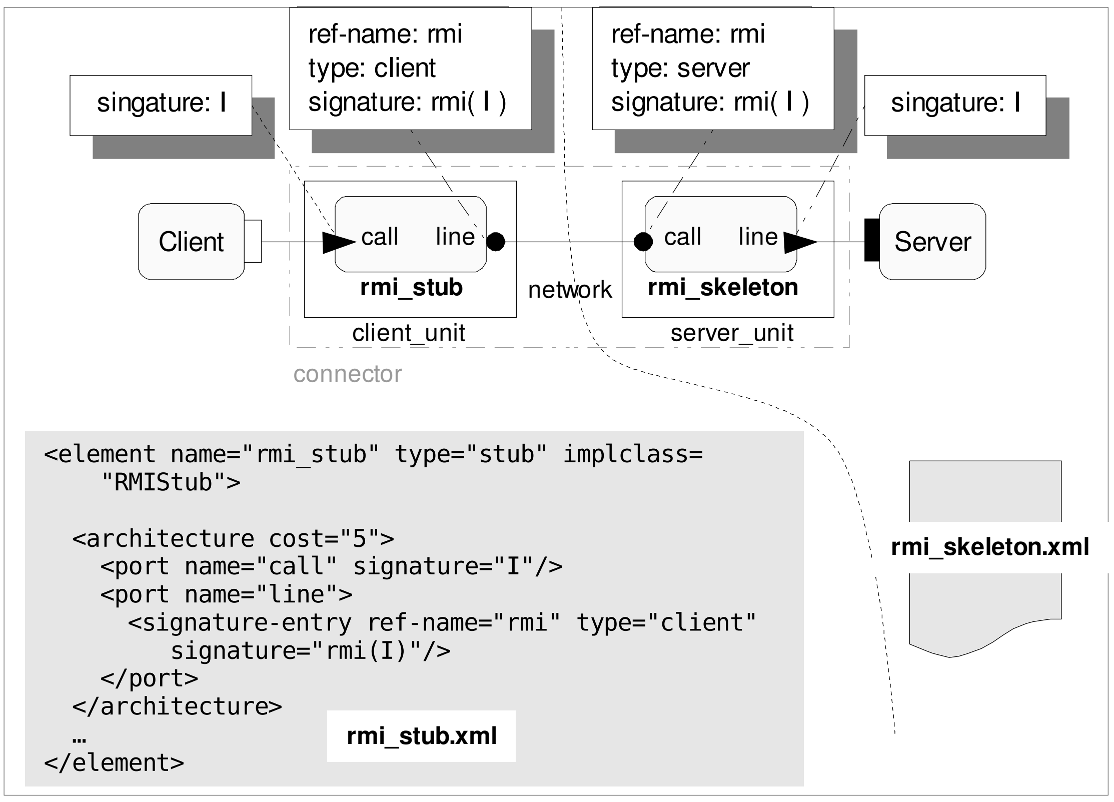


```
<specification>
  <unit name="client_unit" ...>
    ...
    <port name="call" type="provided" signature="java_interface('DemoIFace')"/>
  </unit>

  <unit name="server_unit" ...>
    ...
    <port name="call" type="required" signature="java_interface('DemoIFace')"/>
  </unit>
  ...
</specification>
```

Figure 6.7: A high level configuration of the RMI connector

The client unit contains the element `stub` and the server unit contains `skeleton`. The element configuration files for the stub and skeleton, `rmi_stub.xml` and `rmi_skeleton.xml`, are under the `org/objectweb/dsrg/congen/conrep/elements/` folder (for element configuration files see Chapter 5.3).

Figure 6.8 shows port signatures inside the RMI connector.

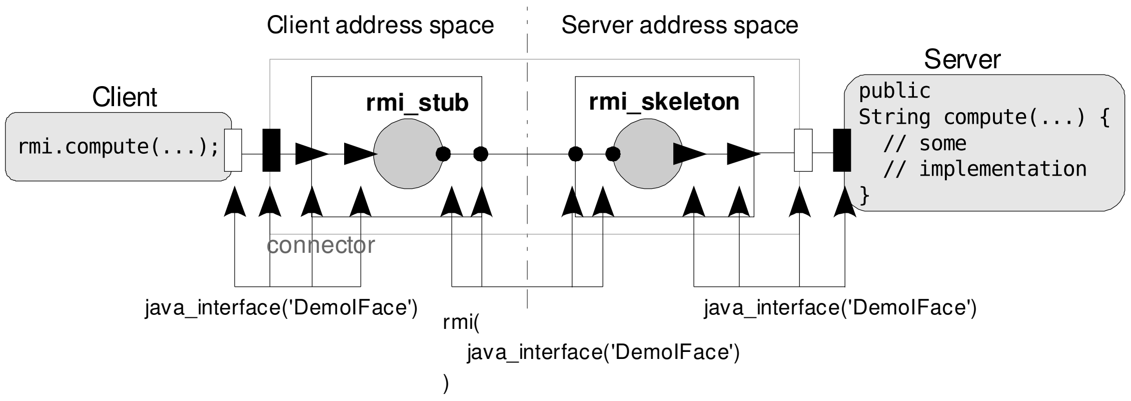


```
<element name="rmi_stub" type="stub" implclass=
    "RMIStub">

  <architecture cost="5">
    <port name="call" signature="I"/>
    <port name="line">
      <signature-entry ref-name="rmi" type="client"
         signature="rmi(I)"/>
    </port>
  </architecture>
  …
</element>
```

Figure 6.8: The signatures of the RMI stub and skeleton

The assigned endpoint signature `java_interface('DemoIFace')` is *unified* with the signature `I` and that with the signature rmi(`I`). The resulting configuration where the all variables `I` have been substituted for `java_interface( 'DemoIFace')` is depicted in Figure 6.9.

The stub and the skeleton are communication between each other through the network using Java RMI. So they utilize the type operator `rmi()` that adjusts the interface `I` in order that it meets Java RMI requirements (see Chapter 6.1).



```
public
String compute(...) {
  // some
  // implementation
}
```

Figure 6.9: A propagation of the DemoIFace interface through the RMI connector

6.2.3 Generating remote interfaces

In order that the stub and skeleton can interoperate, the original `DemoIFace` must be slightly adjusted to meet the RMI requirements – the interface must inherit from the `java.rmi.Remote` interface and each method have to throw the `java.rmi.RemoteException` exception.

For this purpose there is a special function in the type system – `rmi(I)`. In fact `rmi(I)` is only an abbreviation for the operators shown in Figure 6.10:

```
ext_add(
      exc_add(
            trans_refs(I),
            java_class('java.rmi.RemoteException')
      ),
      java_interface('java.rmi.Remote)
)
```

Figure 6.10: Equivalent notation for the rmi(I) operator

Here is their list with descriptions:

- `trans_refs(I)` – processes all methods of the interface `I`. Each argument and return value that is of the business interface type (e.g. `DemoIFace`) is replaced by `java.lagn.String` because the business interfaces are handled specially (see Chapter 6.2.6).

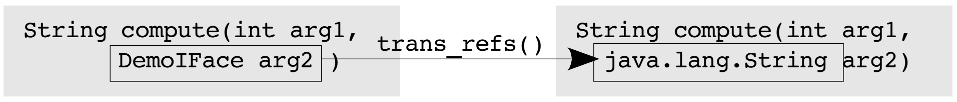


Figure 6.11: Transforming remote interfaces in methods signatures into strings

- `exc_add(I, E)` – to each method's `throws` clause of the interface `I` adds the exception `E`. `rmi()` uses this function for adding `java.rmi.RemoteException` to all methods in the interface in order that the interface can be used with RMI.

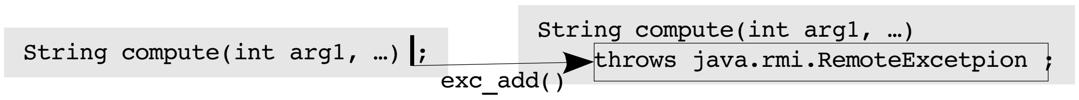

String compute(int arg1, …) ;
exc_add()
String compute(int arg1, …)
throws java.rmi.RemoteExcetpion ;

Figure 6.12: Adding java.rmi.RemoteException to the `throws` *clause of all methods*

- `ext_add(I, E)` – to `I`'s `extends` clause adds the interface `E`. `rmi()` uses it for extending the `java.rmi.Remote` interface. Note that the name of the interface has been changed because of name collisions.

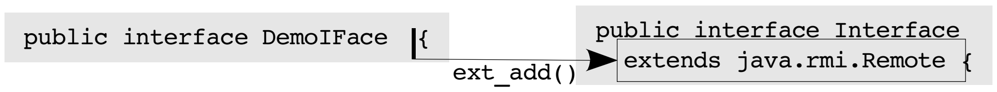
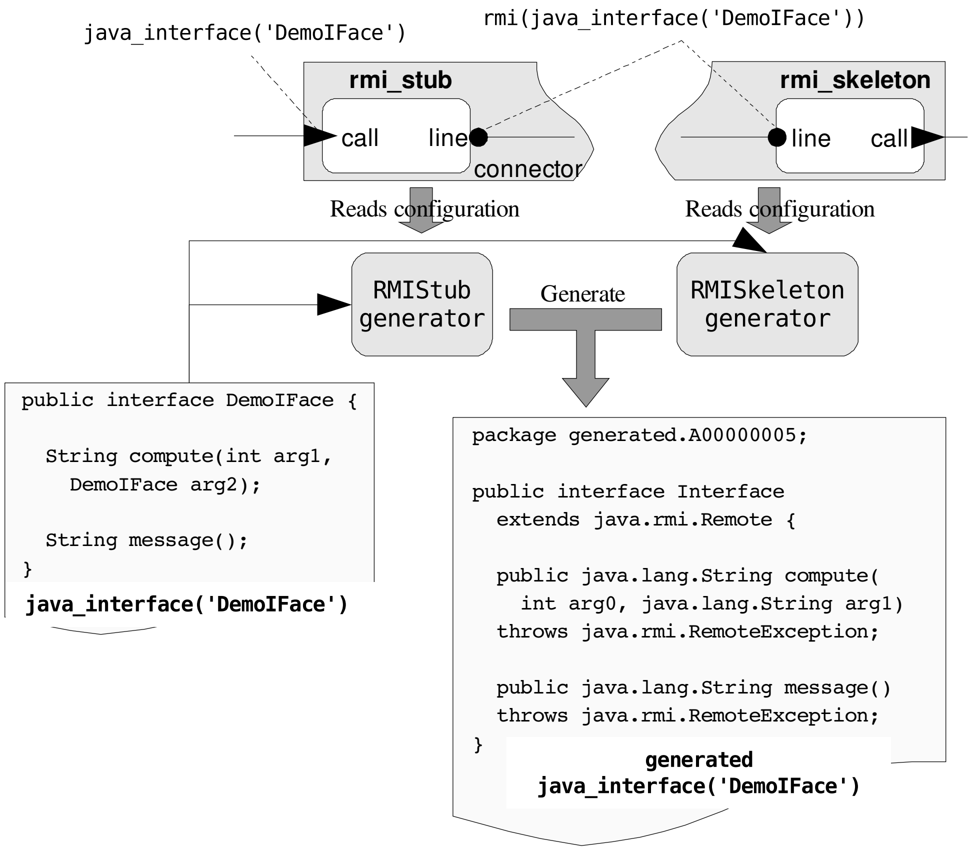

public interface DemoIFace {
ext_add()
public interface Interface
extends java.rmi.Remote {

Figure 6.13: Adding java.rmi.Remote to the `extends` *clause of the interface*

Figure 6.14 shows a process of a generation of some remote interface. The stub as well as the skeleton are attached to application components through the `call` ports. Through them they are given the signature `java_interface('DemoIFace')` and then that signature it is propagated to the `line` ports where the signature `rmi(I)` is substituted by `rmi(java_interface('DemoIFace'))`. The function `rmi()` makes from the originally “non-RMI” `DemoIFace` an remote interface that can be already used by Java RMI. Finally the generated remote interface is written onto the disc and compiled into a binary code.



```
public interface DemoIFace {

  String compute(int arg1,
    DemoIFace arg2);

  String message();
}
```

```
package generated.A00000005;

public interface Interface
  extends java.rmi.Remote {

  public java.lang.String compute(
    int arg0, java.lang.String arg1)
  throws java.rmi.RemoteException;

  public java.lang.String message()
  throws java.rmi.RemoteException;
}
```

Figure 6.14: Generation of the remote interface from the rmi_stub and rmi_skeleton

Although the remote interface is generated by stub as well as by the skeleton, it is created only once because the signature on the server side is the same as on the client side. The generator ensures that identical signatures are compiled only once.

The generated interface is stored into some folder of form “`generated/nnn`” (where `nnn` is sequential number) and always has the name `Interface` (corresponding file name is `Interface.java`, respectively `Interface.class` after the compilation). The names do not collide because they are always in different Java packages.

6.2.4 Templates for RMI

Two main templates used for supporting RMI are the `rmi_stub.template` file and the `rmi_skeleton.template` file located under the `org/objectweb/dsrg/congen/conrep/templates/` folder. From these templates the stub and skeleton, specific for a concrete application, are generated by the code generator (see Chapter 3.3),.

Figure 6.15 shows a stub's structure, the template for the skeleton is very similar.

```
package %PACKAGE%;

/**
 * RMI stub for %TARGET_INTERFACE%.
 */
public final class %CLASS%
    implements
        org.objectweb.dsrg.connector.ElementLocalServer,
        org.objectweb.dsrg.connector.ElementRemoteClient,
        %IMPLEMENTS% {

    /**
     * Local target of business method invocations.
     */
    protected %TARGET_INTERFACE% target;

     ...

    /* ********************************************************************
     * %TARGET_INTERFACE% Methods
     * ********************************************************************/

%TARGET_METHODS%
}
```

Figure 6.15: A structure of `rmi_stub.template`

As it is seen the templates are similar to Java source. But they contain special macros that are replaced by the connector generator to adapt the connector for the specific needs of an application. The stub as well as the skeleton are implemented as one Java class. The following list explains the meaning of the particular macros:

- %PACKAGE% – is a name of a Java package where the resulting code for the stub (or the skeleton) will be generated (e.g. generated.A00000008). The Java language forces that the package name must correspond to the name of the folder. The connector generator ensures that the code will be generated in the appropriate package.
- %TARGET_INTERFACE% - It has a different meaning within the stub and skeleton. In the stub it is a name of the generated RMI remote interface (using the type operator `rmi()` and the auxiliary method `JvavaInterfaceWriter.write()`, see Chapter 5.7 and 6.2.3) from the business interface.

  In the skeleton it is a name of the business interface.

They have in common that the variable `target` is used for forwarding calls. The stub calls the skeleton and the skeleton calls the attached server component as in Figure 6.16:

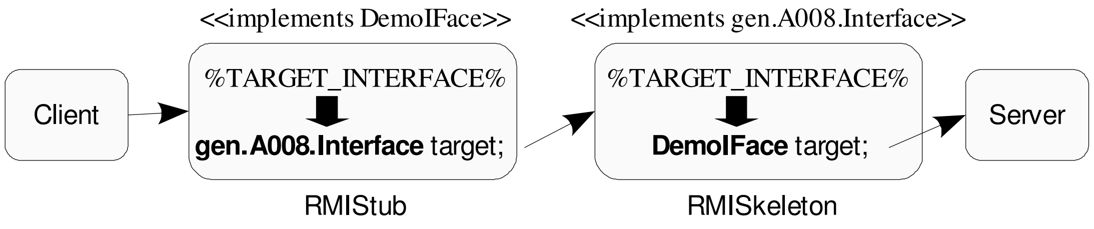


Figure 6.16: Usage of the %TARGET_INTERFACE% macro

- %CLASS% - Is a name of a Java class which the stub of skeleton will be generated into. From a programmer's view the stub or skeleton is only a piece of code encapsulated in one Java class. Its name is taken from the according element configuration file (see Chapter 5.3). For example the stub for RMI is configured in `org/objectweb/dsrg.congen/conrep/elements/rmi_stub.xml`:

```
<script>
  <command action="jimpl">
    . ..
    <param name="class" value="RMIStub"/>
    <param name="template" value="rmi_stub.template" />
  </command>
```

Figure 6.17: Configuration of the name for the class generated from the rmi_stub.template

When more stubs are needed (for various business interfaces) they are distinguished only by the package therefore for example there can be two stubs `generated.a0004.RMIStub` and `generated.a0008.RMIStub`.

The skeleton is configured to have the name `RMISkeleton`.

- %IMPLEMENTS% - Also have different meaning in the stub and skeleton. In the stub it is a name of the business interface (e.g. DemoIFace). The stub implements the business interface because it works as a proxy for the server component which implements the business interface as well.

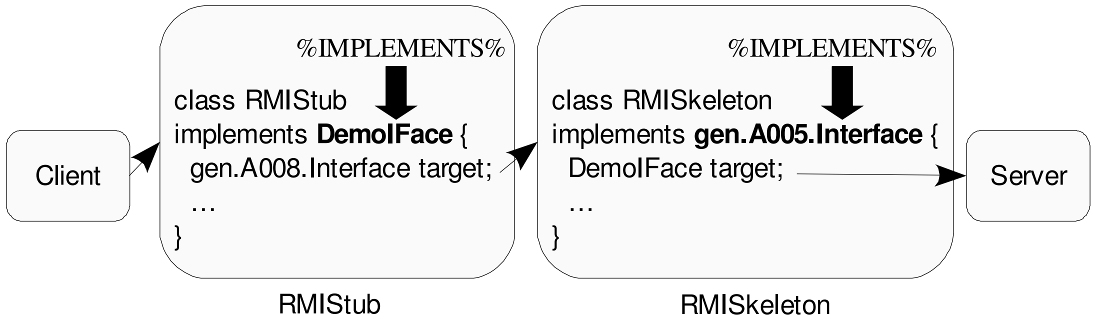


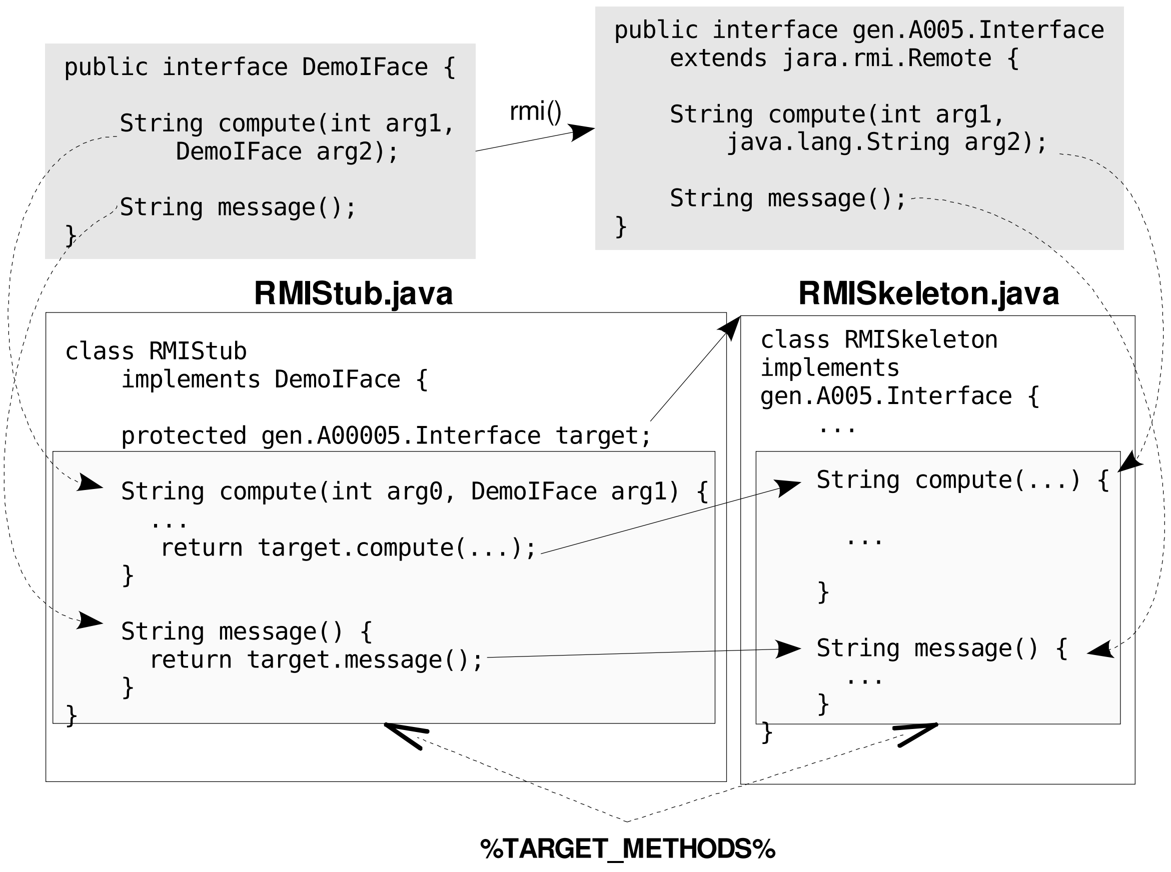
Figure 6.18: A substitution of %IMPLEMENTS%

In the skeleton it is the generated RMI remote interface from the business interface (for example `generated.a0005.Interface`) because it can be used in Java RMI. The `RMISkeleton` class register itself with `rmiregistry` so the `RMIStub` object can call it, see Figure 6.18.

- %TARGET_METHODS% - for each method from the business interface is generated corresponding method in the `RMIStub` class (and also in the `RMISkeleton` class) with the same name (Figure 6.19).

  In `RMIStub` arguments and return values are the same as in the business interface, but in `RMISkeleton` they are changed in dependence on the type of the argument.

  - Primitive values are left unchanged.
  - Business interfaces are changed to `java.lang.String` (for the reason explained later). So that for example the method "`String compute( int arg1, DemoIFace arg2)`" from `DemoIFace` is changed to "`String compute(arg1, java.lang.String arg2)`".
  - Objects or non-business interfaces are not supported.



```
public interface DemoIFace {

    String compute(int arg1,
        DemoIFace arg2);

    String message();
}
```

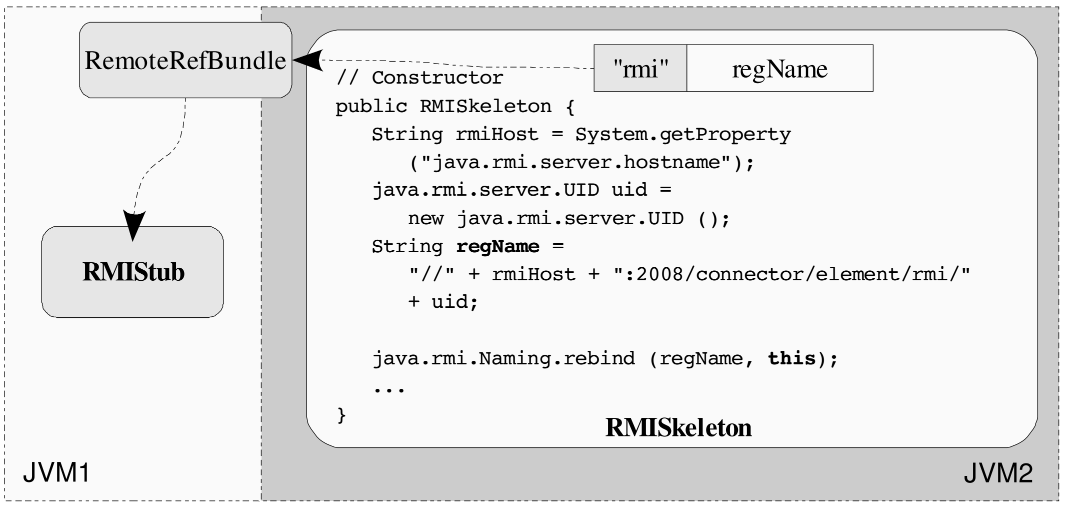
```
public interface gen.A005.Interface
    extends jara.rmi.Remote {

    String compute(int arg1,
        java.lang.String arg2);

    String message();
}
```

```
class RMIStub
    implements DemoIFace {

    protected gen.A00005.Interface target;

    String compute(int arg0, DemoIFace arg1) {
      ...
       return target.compute(...);
    }

    String message() {
      return target.message();
    }
}
```

```
class RMISkeleton
implements
gen.A005.Interface {
    ...
    String compute(...) {

      ...

    }

    String message() {
      ...
    }
}
```

Figure 6.19: A connection between business interface and %TARGET_METHODS%. `RMIStub` and `RMISkeleton` contain all methods from the `DemoIFace` interface.

6.2.5 Remote bindings



```
// Constructor
public RMISkeleton {
   String rmiHost = System.getProperty
      ("java.rmi.server.hostname");
   java.rmi.server.UID uid =
      new java.rmi.server.UID ();
   String regName =
      "//" + rmiHost + ":2008/connector/element/rmi/"
      + uid;

   java.rmi.Naming.rebind (regName, this);
   ...
}
```

Figure 6.20: Scheme of the remote binding

In order that the stub can communicate with the skeleton it must obtain the skeleton's RMI address (see Chapter 5.4).

Figure 6.20 shows a piece of code taken from the `RMISkeleton`'s constructor. It gets a name of the host where the `rmiregistry` server runs. The property `java.rmi.server.hostname` can be set by the `java`'s `-D` option, e.g. `-Djava.rmi.server.hostname=rmi.server.host.com`.

The variable `uid` represents an identifier which is unique within the host it is generated on.

The string `regName` is created from `rmiHost` and `uid` in order to form an unique URL address (without the protocol and colon at the beginning). The domain part of the address, created from rmiHost, with the port number 2008 is pointing to the running `rmiregistry` service.

`java.rmi.Naming.rebind()` then associates the string `regName` with the `RMISkeleton` object. regName is passed to the global “bundle” of references, extracted by the `RMIStub` which use it with the function `java.rmi.Naming.lookup()` to obtain the RMI reference to RMISkeleton.

6.2.6 Argument passing

In the stub and skeleton is for each method from the business interface generated a function with the same name (see Chapter 6.2.4).

A body of such a function in the stub prepares arguments for the corresponding method in the skeleton and calls it (see Figure 6.19). The skeleton's function does the similar thing – prepares arguments and calls the server component implementing a business logic.

Arguments are processed differently based on their type. Primitive types are left unchanged and just passed through. It becomes a little bit complicated when passing a remote reference.

In the stub is the remote reference changed to the string ID using the function `DockConnectorManager.getConnectorUnitByReference()`. It is because the remote reference is just a local reference to some stub and it cannot be passed through the network. When it is transformed to the ID, which is `java.lang.String`, it can be without problems handled with Java RMI.

The skeleton must do a reverse operation – from the ID create the remote reference. It is done by the function `DockConnectorManager.createSharedConnectorUnit()` which based on the ID creates a local `RMIStub` associated with the same server as the original remote reference from the client component.

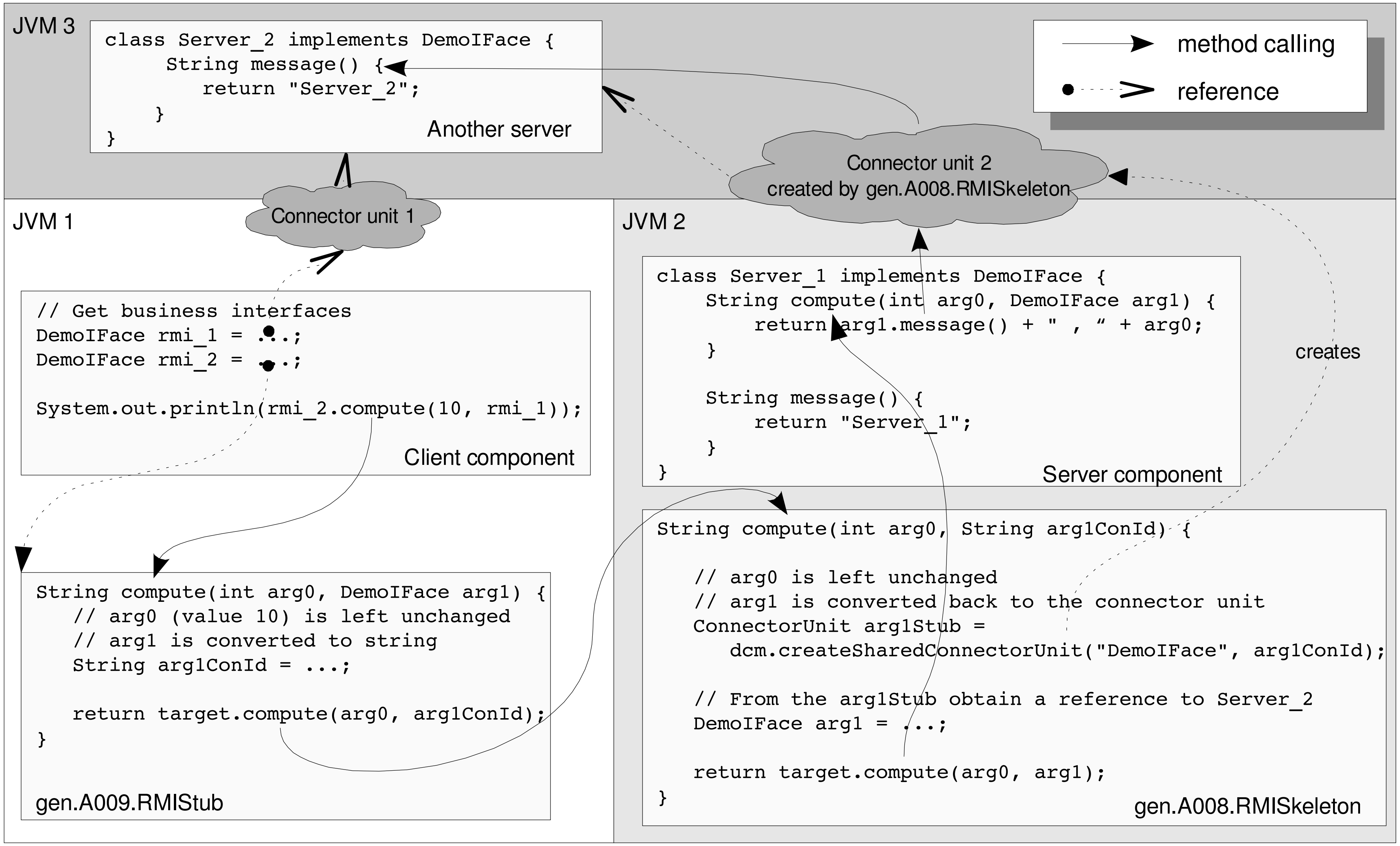
Passing objects through the connector in not supported.

Figure 6.21 shows an example where a remote reference is passed. There are three address spaces – one clients and two servers. The client is holding two remote references (of the `DemoIFace` type) in the variables `rmi_1` and `rmi_2`. Through the variable `rmi_2` calls the function `compute()` where passes two arguments – the first is the primitive type (value 10) and the second is the remote reference `rmi_1`.

The call is mediated through `get.A009.RMIStub` which left the first argument unchanged (value 10) and from the second argument (`rmi_1`) gets the ID (it is ID of the connector unit, in the picture with the name “Connector Unit 1”). Thus the skeleton's function `compute()` is called with one integer and one string argument. The class `gen.A008.RMISkeleton` creates from the string a local connector unit (in the picture “Connector Unit 2”) and calls the server with appropriate parameters.

The server's body calls `arg1.message()` which through the created “Connector unit 2” invokes the `message()` method on the other server `Server_2` and returns the string “Server_2”. That string is concatenated with the value 10 and returned to the client. Therefore the function `System.out.println()` in the client prints the string “Server_2 , 10”.

The client can also pass the variable `rmi_2` instead of `rmi_1` as a second parameter in the `compute()` function. In that case the string “Server_1 , 10” would be printed.



```
class Server_2 implements DemoIFace {
     String message() {
        return "Server_2";
     }
}
```

```
// Get business interfaces
DemoIFace rmi_1 = ...;
DemoIFace rmi_2 = ...;

System.out.println(rmi_2.compute(10, rmi_1));
```

```
String compute(int arg0, DemoIFace arg1) {
   // arg0 (value 10) is left unchanged
   // arg1 is converted to string
   String arg1ConId = ...;

   return target.compute(arg0, arg1ConId);
}
```

```
class Server_1 implements DemoIFace {
    String compute(int arg0, DemoIFace arg1) {
        return arg1.message() + " , " + arg0;
    }

    String message() {
        return "Server_1";
    }
}
```

```
String compute(int arg0, String arg1ConId) {

   // arg0 is left unchanged
   // arg1 is converted back to the connector unit
   ConnectorUnit arg1Stub =
      dcm.createSharedConnectorUnit("DemoIFace", arg1ConId);

   // From the arg1Stub obtain a reference to Server_2
   DemoIFace arg1 = ...;

   return target.compute(arg0, arg1);
}
```

Figure 6.21: A scheme of passing of remote references

6.2.7 Summary

From the previous chapters results that there exists partial support for Java RMI. It can handle primitive data types (and the the type `java.lang.String`) and also there are some mechanisms for passing remote references.

But support for passing objects is not implemented at all. These are two major difficulties:

- In order that an object can be passed through the network it must be marked as `java.io.Serializable` (it must implement `java.io .Serializable` interface). Otherwise the runtime exception will be thrown. We want to explore this issue and discuss whether this limitation could be bypassed.
- Objects in general can also contain remote references inside them. These references should be processed in a similar way as arguments of business interface type of remote methods (see Chapter 6.2.6), that is to each remote reference on the client side generate a proper string ID which is then used on the server side for creation of corresponding local connector unit.

So we want to aim on the previous problems and implement the full support for object types within Java RMI.

6.3 Improved support for RMI

6.3.1 Dealing with `java.io.Serializable`

In order that an object type can be used as an argument in a remote function it must implement the `java.io.Serializable` interface. Fortunately that interface does not contain any function. Its purpose is only mark the type as “ready to serialize” and Java then handles a serialization process automatically.

One of the objectives was to bypass this limitation in order that programmer would not be bothered by the duty to mark each object he wants to use within the connector by `java.io.Serializable`. But it seems that Java designers had a good reason that object are not serializable by default and users must mark them them explicitly.

Here are two arguments which it could have been:

- Some object are closely tied with the context of the Java Virtual Machine or a computer where the program runs. For example `java.io.File` is associated with the OS filesystem, it would not have any meaning if it was transferred to an other computer. In general any object connected with an OS resource is a nonsense to serialize (threads, sockets, opened database connection, ...).
- In Java when an object is passed to an ordinary function, it is passed by reference (in contrast to primitive types which are passed by value). It means that if a state of the object is changed (e.g. some field) in the body of the function, the change is also visible after the return from that function. But in RMI it is not the case. When an object is passed to a remote server, the object is first serialized and then sent through the network. If it is changed on the server (either explicitly or by a side effect), the changes are not reflected back to the client. Even if it was implemented that all objects were sent back to the client it would not avoid the kind of side effects like writing to a file. Such side effects are almost impossible to mirror on the client side.

Unfortunately the conclusion is that the programmer always must be aware of that he is using a remote method and thus be careful what parameters he is passing. It seems that RMI will never be fully transparent, but a little extra care is worth the power of remote objects.

6.4 Addressing support for object types

The previous chapter gives the reason why the use of `java.io.Serializable` is inevitable. But that is still not sufficient for supporting general objects which can be quite complex. If it was restricted that they could not contain inner remote references the Java engine would take care about all needed things to transport the object across the network. But in order that the connector would be able to support also nested remote interfaces (and that is desired), extra attention must be paid to the serialization and the deserialization process and treat remote references differently.

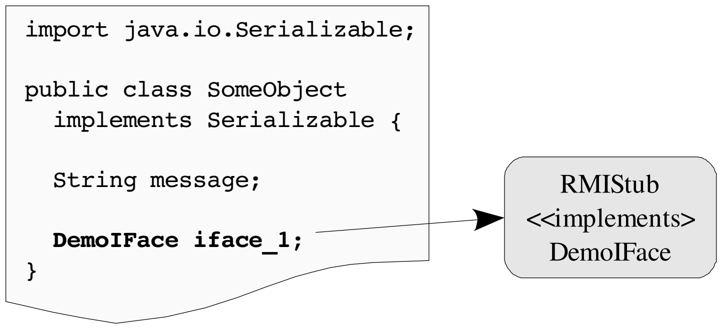

```
import java.io.Serializable;

public class SomeObject
  implements Serializable {

  String message;

  DemoIFace iface_1;
}
```

Figure 6.22: Some object containing a remote interface (the variable remote). It must be handled separately, the default Java routines is not possible to use.

The java language allows a developer slightly modify the process of serialization/deserialization using the special functions inside a serializable object:

```
private void writeObject(java.io.ObjectOutputStream out)
      throws IOException

private void readObject(java.io.ObjectInputStream in)
      throws IOException, ClassNotFoundException;
```

The `writeObject()` function is responsible for writing the inner state of the object to the output stream so that the opposite function `readObject()` can restore it on the other side. The programmer has also at disposal the default function `out.defaultWriteObject()` which invokes the standard serialization mechanisms provided by the Java engine.

But the imagination that the user has to define these function for each object he wants to use within the connector is inacceptable. The process should be most transparent as possible.

6.4.1 Basic Idea

Finally we have used the Java Reflection [33]. It is a very powerful technology that allows at runtime introspect objects and even modify them. The main idea is before sending an object to the server go recursively through it and change all remote references appropriately (how it will be discussed later). The first thing which is needed is to distinguish between remote and local references and for that purpose it seems that *Java Annotations* [34] are the most eligible.

Java Annotations are a way how to add some extra information (metadata) to a Java source. Some can be then available to a programmer at runtime.

We use it to annotate the `RMIStub` classes. All remote references all pointing to `RMIStub` and if all `RMIStub` classes are marked with some special annotation we can distinguish remote references from the others.

The annotation used for marking stubs is defined in the `org/objectweb/dsrg/connector/rmi/ConnectorInterface.java` file:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ConnectorInterface {
       String value();
}
```

Figure 6.23: Definition of the annotation used for marking stubs

and here is a description of that annotation:

- `@Retention(RetentionPolicy.RUNTIME)` makes the annotation information available at runtime. If it was not used the annotation could be read only from sources.
- `@Target(ElementType.TYPE)` – says that the annotation is applicable only to classes. The other possible value is for example `@Target(ElementType.METHOD)`.
- `String value()` - declares the the annotation will have one string property. That property designates which wrapper (see Chapter 6.4.2) should be used for the associated `RMIStub` class.

The template `rmi_stub.template` is changed appropriately to use that annotation:

```
@org.objectweb.dsrg.connector.rmi.ConnectorInterface("%WRAPPER%")
public final class %CLASS%
    implements
  ...
```

Figure 6.24: Adjusted `rmi_stub.template`

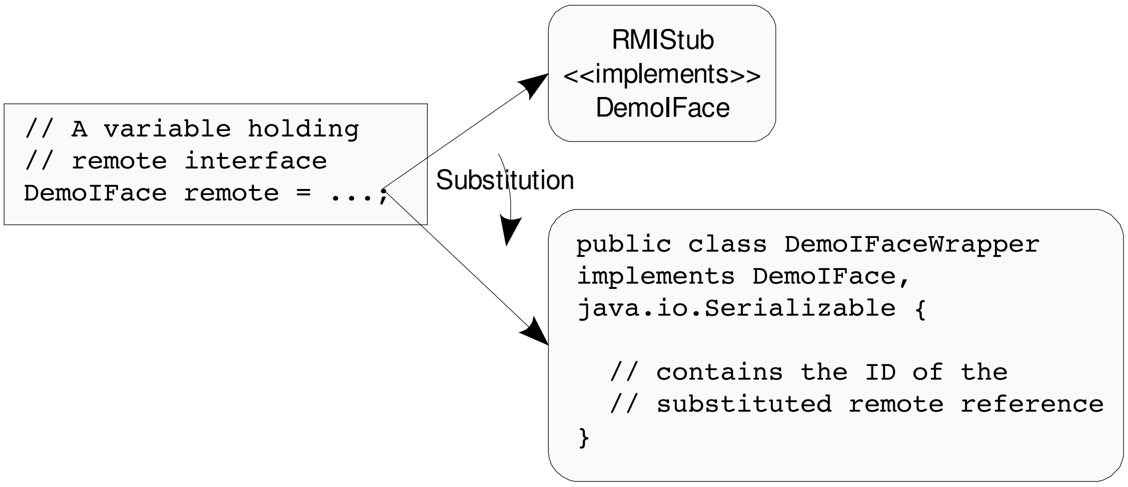
So when the client component has two references e.g. of the type `DemoIFace`, and one is referring to the stub and the other just to some local object accidentally implementing the interface `DemoIFace`, the connector can easily recognize them at runtime by checking the annotation via the function `isAnnotationPresent(ConnectorInterface.class)` of the `Class` object.

6.4.2 A wrapper

Now when it is possible to identify which references are remote and which are not, the remote ones can be replaced by some special information so that the server based on than information can create a local connector unit. But the original reference cannot be substituted by a whatever object type. The Java language forces that variable of a type `A` can hold only object of a type `A` or a type derived from `A`.

Remote references can be held only in variables of a business interface type, thus there there is for each business interface generated a special class, a wrapper, that is capable of holding some information about the remote reference. Besides the information about the reference, which is useful on the server side, the wrapper must implement the business interface (even it has no functionality) in order that the wrapper can be hold in the variable of a business interface type.

Figure 6.25 shows how the remote interfaces are replaced. The variable `remote` is originally holding a reference to the `RMIStub`. But in order the variable could be transported over the network it is replaced by the appropriate wrapper class which holds the ID associated with RMIStub.



```
// A variable holding
// remote interface
DemoIFace remote = ...;
```

```
public class DemoIFaceWrapper
implements DemoIFace,
java.io.Serializable {

  // contains the ID of the
  // substituted remote reference
}
```

Figure 6.25: Replacing remote interface with the appropriate wrapper

In the type system accrues the new operator `add_rmi_wrapper(I)` which to the business interface `I` produces a suitable wrapper.

The connector generator uses that type to prepare all wrappers which will be needed at runtime. Here is a piece of code from the `RMIStub` generator:

```
RMIInterfaceWrapper wrapper = (RMIInterfaceWrapper)typeFactory.getType(
    new SymbolicTypeSpecifier("add_rmi_wrapper", serverJavaIface)
);

wrapper.compile(javaTools);
```

Figure 6.26: A fragment of code from the RMIStub generator

The variable `wrapper` holds an internal description of the wrapper. The expression (after the type cast operator `(RMIInterfaceWrapper)`) is simply programmatic representation of a symbolic type. If the variable `serverJavaIface` represents the `DemoIFace` interface, the whole expression can be imagined as the signature "`add_rmi_wrapper(java_interface('DemoIFace'))`".

The function `wrapper.compile()` assures that the internal representation of the wrapper will be written onto the filesystem as a Java source and then compiled into a Java class (e.g. `generated.A00000009.Wrapper`).

`RMIStub` acts as a server to the client component and only references to RMI stubs are interpreted as remote so when each stub generates its associated wrapper it will be guaranteed that at runtime the connector will have all necessary wrapper classes available.

In order that the server can create an according stub (to the client's stub) in its address space it needs only two information – the id associated with the stub and the type of the business interface.

Here is an example of a wrapper generated for `DemoIFace`:

```
@org.objectweb.dsrg.connector.rmi.ConnectorInterfaceWrapper("DemoIFace")
public class Wrapper
    implements DemoIFace, java.io.Serializable {

    public String connectorId;

    public java.lang.String compute(int arg0, DemoIFace arg1) { return null; };
    public java.lang.String message() { return null; };
}
```

Figure 6.27: A wrapper generated for DemoIFace

The next annotation used by the connector is `org.objectweb.dsrg .connector.rmi.ConnectorInterfaceWrapper.` Its purpose is similar as `@ConnectorInterface` to annotate stubs. While the `@ConnectorInterface` is used within the serialization in order the connector could recognize "ordinary" references from remote remote references, the `@ConnectorInterfaceWraper` is used in the deserialization to distinguish between "ordinary" objects and wrappers. It has one string property (in the picture “`DemoIFace`”) that says which business interface the wrapper is related to.

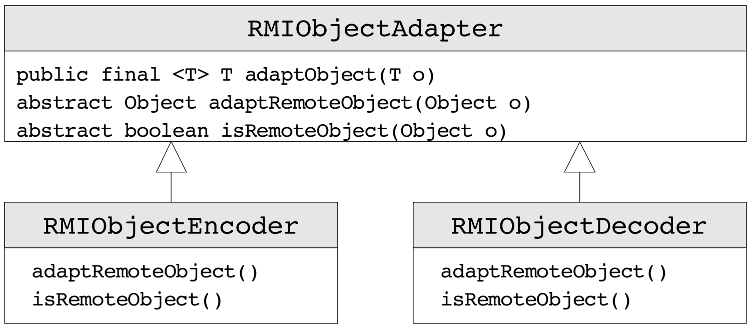
Here are some notes for Figure 6.27:

- The wrapper implements `java.io.Serializable` because it is transferred across the network using Java RMI and RMI requires that.
- The field `connectorId` contains the id of the connector which the wrapper belongs to. From this information and from the property of `@ConnectorInterfaceWraper` the server is able to create a local stub in its address space.
- The rest of the body contains only empty implementations of the functions from the business interface. They are doing nothing but they must be there to satisfy the “contract” of the class (implement all interfaces stated after `implements` keyword).
  Each function contains only the return statement. Return value depends on a return type of the function. For boolean type `false` is used, `0` is used for primitive types except boolean and `null` for object types.

6.4.3 RMIObjectAdapter

We have implemented the class `RMIObjectAdapter` that is used for adjusting objects before they are passed to a remote function (thus before the serialization) on the client side a then just before the server business logic takes control (just after the deserialization). It goes recursively through the object and searches for the remote references (recursively because the object can contain subobjects). Before the object is passed to the server all remote references are replaced by the appropriate wrappers and then on the server side the wrappers are back replaced by suitable local connector units.

`RMIObjectAdapter` is extended by the two subclasses `RMIObjectEncoder` and `RMIObjectDecoder` (see Figure 6.28):


RMIObjectAdapter
public final <T> T adaptObject(T o)
abstract Object adaptRemoteObject(Object o)
abstract boolean isRemoteObject(Object o)
RMIObjectEncoder
adaptRemoteObject()
isRemoteObject()
RMIObjectDecoder
adaptRemoteObject()
isRemoteObject()
Figure 6.28: Class hierarchy of the RMIObjectAdapter

The main method of `RMIObjectAdapter` is `adaptObject(T o).` It has one parameter of an object type and it is supposed to return the same object with replaced internal remote references. It goes through the object field by field and processes differently various types:

- **Primitive types** are ignored because they cannot be assigned an object type and thus not even a remote reference.
- **Object types (except arrays)** – when `RMIObjectAdapter` finds a field of an object type it calls the method `isRemoteObject(Object o)` which have to decide whether the object is associated with a remote reference (remote object or wrapper). If it returns `true` the function `adaptRemoteObject(Object o)` is called on that object. In case the object is not remote the whole processing is repeated recursively on that subobject. Both functions `isRemoteObject()` and `adaptRemoteObject()` are supposed to be overrideen in subclasses.
- **Arrays** – whether the array will be processed or not depends on the *component type* of an array, e.g. `'int[]'` has the component type `'int'` and `'String[][][]'` has the component type `'String'`.
  - If the component type if primitive, the array is ignored because there is no way how into such an array include a remote object. E.g. the array `int[][][]` in the leaves must contain only integers (otherwise a compile time will be thrown) and nodes of the array can be either `null` or an array of integers of a corresponding dimension (e.g. the first dimension can contain only `int[][]`, the second `int[]` etc.). Here is an example:

```
int[][] a = new int[2][];
a[0] = new int[2];
a[0][0] = new Object();      // Compile time error!
a[1] = (int []) new Object(); // Runtime error
```

- If the component type is of object type the whole array is traversed and all leaves processed like objects.

An extra care must be paid because of possibility of circular references. An object can contain (directly or inside some of its subobjects) reference to itself. Some circular references can be quiet subtle, even if they are not written explicitly in a source code. For example Java Inner Class holds hidden pointer made by the compiler to the parent class.

The `RMIObjectAdapter` class must deal with that. That is why it contains field of the type `Map<Object, Object>`[1] (from the Java Collections[2] [35]) of already processed, or just being processed, objects.

```
private Map<Object, Object> beingProcessed =
  new IdentityHashMap<Object, Object>();
```

Naturally it seems that the standard set implementation `HashSet<E>`[3] would be appropriate for storing *set* of objects. But it compares objects using the inherited function `boolean equals(Object obj)` from `java.lang.Object`. Instead the comparison of objects based on the equality of references (whether two references point at the same objects) is needed. The function `equals()` from the `java.lang.Object` class exactly do that but that implementation can be overridden in subclasses.

Standard Java libraries do not contain an implementation of the set that would compare objects by reference equality. That is why we used `IdentityHashMap<Object, Object>()`. In fact the mapping is not needed, but it was one collection which compare references. Only keys from the map (by the `put()` and `get()` functions) are used, the actual values of the mapping are insignificant.

The concrete behavior of the `RMIObjectAdapter` class is then refined in its subclasses. Here is a list of that two subclasses and its functions:

- `RMIObjectEncoder` – is supposed to replace all references in some object by their appropriate wrappers. It is achieved by overriding this two functions:
  - `isRemoteObject()` - detects remote references. As has been said only references to connector's stubs are regarded as remote and the stubs are marked by the special annotation (see Chapter 6.4.1). Thus it is enough to check if the object contains that annotation by this function of the `java.lang.Class` class:

1 A Map is an object that maps keys to values.

2 A Collection represents a group of objects.

3 A set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The HashSet is one concrete implementation of set.

```
   isAnnotationPresent(ConnectorInterface.class)
```

  - `adaptRemoteObject()` - from the object annotation property discovers which wrapper should be used for that object, instantiate the wrapper and set its field `connectorId` to the proper value.

- `RMIObjectDecoder` – it works in contrast to `RMIObjectEncoder`. It goes through the object and replace all wrappers by appropriate connector units. It also overrides the two functions"
  - isRemoteObject() - detect if an object is a wrapper. All wrappers are marked by a special annotation, this function check it by using function

```
   isAnnotationPresent(ConnectorInterfaceWrapper.class)
```

  - `adaptRemoteObject`() - from the wrapper extracts the id of the connector an from the annotation's property a name of the business interface. This information is enough for creation of local connector unit. The wrapper is replaced by that unit.

`RMIObjectEncoder` and `RMIObjectDecoder` are used on both sides (client and server). For an explanation how they are used in `RMIStub` and `RMISkeleton` see the next chapters.

6.4.4 Integration with the connector

Besides of implementing the `RMIObjectAdapter` class and all stuff connected with it (encoder, decoder, annotations...) there were also some adjustments in the original sources.

The previous implementation was changing signatures of methods in RMI remote interfaces (see Chapter 6.2.3) and with it also connected method signatures in the `RMISkeleton` class (see Chapter 6.2.4 and 6.2.6). Concretely the arguments of a business interface type were changed to `java.lang.String`.

But since `RMIObjectEncoder` transforms inner remote references inside objects as well as “standalone” remote references (not nested) to their corresponding wrappers, now it is possible to pass object arguments through the network (but they still must be marked as serializable). It is even not necessary to change the method signatures in `RMISkeleton` anymore. When passing the remote reference it is substituted by the wrapper implementing the same business interface. And “ordinary” objects are not changed to another type, only some of their fields, if they are remote references, can be substituted.

Figure 6.29 shows a situation where the `RMIStub` class is passing a remote reference. The function `adaptObject()` before sending that remote reference to the server converts it to the appropriate wrapper also implementing the `DemoIFace` interface. The skeleton's function `compute()` has parameters of the same type like stub's `compute()`.

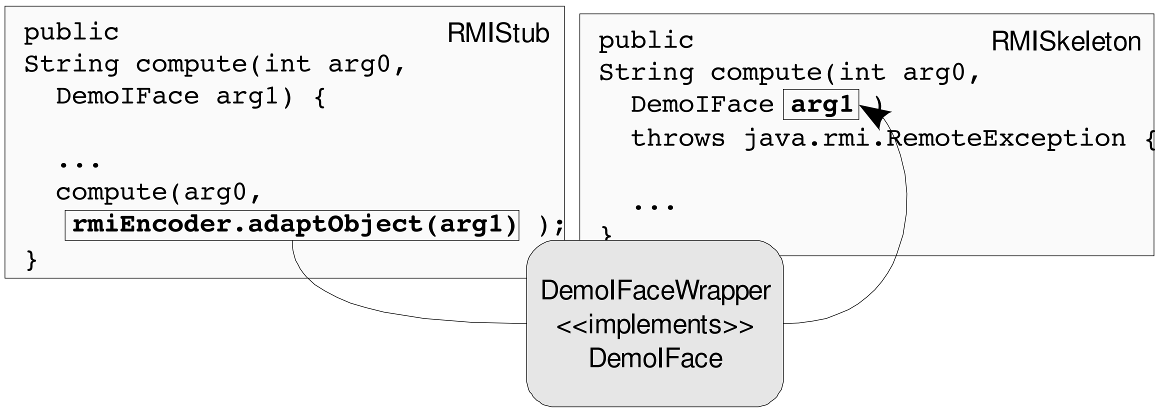


Figure 6.29: A remote reference passing. The stub's methods have the same arguments as skeleton's methods.

RMI remote interfaces were created through the special type operator `rmi(I)` (see Figure 6.10) which was using `trans_refs(I)` to change business interfaces in arguments to strings. Now this function is omitted so the new operator `rmi(I)` has this semantics:

```
ext_add(
      exc_add(
            I,
            java_class('java.rmi.RemoteException')
      ),
      java_interface('java.rmi.Remote)
)
```

Figure 6.30: Modified semantics of the `rmi(I)` type operator

Stub and skeleton generators besides the generation of the RMI remote interfaces created from business interfaces (vie the `rmi(I)` operator) in addition generate also wrapper classes. It is achieved by the new type operator `add_rmi_wrapper(I)` (see Chapter 6.4.2).

The generated code of the stub and skeleton has been changed too. The `RMIObjectEncoder` and `RMIObjectDecoder` are used to adapt function arguments and return values (if they are of object type). Their occurrence in source code is mirrored in the client and server. Whereas `RMIObjectEncoder` is called on the client side to adjust function parameter before sending them to the server and `RMIObjectDecoder` to transform back the return value (from a wrapper), on the server side it is exactly opposite – `RMIObjectDecoder` changes the input parameters (objects that can contain wrappers or are wrappers themselves) to “normal” objects and `RMIObjectEncode` is called before the return value is sent back to the client.

Here are for example two pieces of code from the `RMIStub` and `RMISkeleton` classes.

- RMIStub

```
01 public java.lang.String compute (int arg0, DemoIFace arg1) {
02   ...
03   try {
04     result = this.target.compute (arg0, rmiEncoder.adaptObject(arg1));
05     ...
06     return rmiDecoder.adaptObject(result);
07   } catch (java.rmi.RemoteException) {
08     ...
09   }
10 }
```

Figure 6.31: Structure of compute() from the RMIStub.
The function generated from the DemoIFace interface.

The function `compute()` (line 1) is called from the client component. The first parameter is of type `int` and the second is interface `DemoIFace` which must be adapted first before it can be sent to the server. If it is a remote reference (annotated with `@ConnectorInterface`) the function `rmiEncoder.adaptObject(arg1)` returns a completely new object (the wrapper holding information about the reference). Otherwise the argument `arg1` is just a "normal" local object implementing `DemoIFace`. In such a case the encoder returns the same object but possibly with some changed fields (if it contains some inner remote references).

After the server function returns (line 4) its result must be converted back (notice that `java.lang.String` is also object and therefore it is adapted). It is done on line 6 by the function `rmiDecoder.adaptObject(result)`. After the object is transformed it is returned to the client component which have invoked the `compute()` function.

- `RMISkeleton`

```
01 public java.lang.String compute (int arg0, DemoIFace arg1)
01     throws java.rmi.RemoteException {
03  ...
04  try {
05    result = this.target.compute(arg0, rmiDecoder.adaptObject(arg1));
06    ...
07    return rmiEncoder.adaptObject(result);
08  } catch (RMIObjectAdaptorException e) {
09    ...
10  }
11 }
```

Figure 6.32: The compute() function taken from RMISkeleton.

The function `compute()` (on line 1) located on the server is called from the client's stub. It is inside some object which is registered with `rmiregistry`. The second parameter of the function has the type `DemoIFace` but it is "encoded" in the way described earlier. So before `RMISkeleton` calls the "real" server's `compute()` function (`RMISkeleton`'s compute acts only as a mediator) it must decode the `DemoIFace` object. On line 5 the server's function is called via the `target` field. Primitive type parameters are just passed, there is no need to adjust them in any way, but the object type parameters must be first modified by the `rmiDecoder.adaptObject()` function.

On line 7 the return value is sent back to the client. Again if it is of a primitive type it is leaved unchanged, otherwise `rmiEncoder.adaptObject()` is used.

So here is a brief scenario how the extended support for RMI is used:

Deployment time:

1. To the business interfaces the appropriate stubs and skeletons are generated (which internally use `RMIObjectEncoder` and `RMIObjectDecoder`). For the each interface the wrapper is also prepared (using the new type operator `add_rmi_wrapper`).

Runtime:

2. The client holds the remote interface (reference to local `RMIStub`) and calls some function on it.
3. Corresponding function of the `RMIStub` is invoked. Before it calls the matching `RMISkeleton`'s function it must modify all object parameters via `RMIObjectEncoder.adaptObject()`. It internally uses the prepared wrappers from Step 1. Primitive types are left unchanged.
4. `RMISkeleton` decodes object parameters through the `RMIObjectDecoder .adaptObject()` method and forwards the calling to the server component's function. After the server returns some value, it must be back encoded if it is an object.
5. `RMIStub` decodes the return value (if it is of an object) and passes it back to the client component.

6.4.5 Summary

We have extended the connector's RMI support by the object types. Here are some notes which argument types are actually possible to use and what are the restrictions:

- **Primitive types**

  Primitive types, that is **byte**, **short**, **int**, **long**, **float**, **double**, **boolean** and **char**, are fully supported.

- **Objects**

```
public class MyParentClass {
 public String message() { ... }
}

public class MyClass extends MyParentClass implements
 java.io.Serializable, SomeInterface {
    public static int staticField; // Wrong

    public static final double PI = 3.141592653589793; // Ok
    public static final int random =
          (int) (Math.random() * 100); // Wrong

    // All access modifiers can be used
    public int publicField;
    protected int protectedField;
    private int privateField;
    int field;

    // Object fields must be also serializable!
    public MyOtherClass objectField;


    // Methods
    public static int someStaticMethod() { ... }// Unusual,but Ok
    public String message() { ... } // Ok

    // Nested and inner classes
    static class NestedClass { ... } // Ok
    class InnerClass { ... } // Ok
}
```

There are some subtle aspects which a user should be aware of and avoids them. The piece of code above demonstrates which is possible to use without any troubles and which is not recommended.

A class can inherit from some other classes and implement any number of interfaces. Classes which are supposed to be used in the remote commutation must at least implement `java.io.Serializable` (it contain no methods, it is just a mark). There would be no compile time error but if objects not conforming to this rule were used in the connector, a runtime would error occur.

Static fields should not be used at all (fields with `static` keyword). One exception is using them together with the `final` keyword but it can be dangerous too (see later). It is because static fields are something like global variables known from other languages. It is expected that all objects of the same type share that fields. When one object changes the value of the static field, another object immediately sees that modification. It is true if an application runs in one address space but it is not the case with RMI. The RMI application is inherently distributed where the server and clients run in the different Java Virtual Machines and mostly on different computers. Changes of static fields on a client are not reflected to the server and vice versa. They are ignored by the serialization, only instance fields are transferred through RMI. Each JVM holds its own copy of static fields.

But when the static field is used also with the `final` keyword it can be safe. It depends on which expression is used to initialize the field. If a constant expression is used (like 3.141592) there is no problem. The field is assigned always the same value and the `final` keyword assures that this value will not be changed later. It does not matter that there are more copies of the same field (for each JVM), they all have the same value and cannot be changed.

The other case is a static final field with non-constant initializer. Here it is a problem, it is demonstrated in the example below (it uses `MyClass` from the previous example):

`Client.java`

```
public class Client {
   public static void main(String[] args) throws Exception {
         ...
         // Get the remote interface
         TestConstants testConstants = ...;

         System.out.println(testConstants.test(MyClass.j));
   }
}
```

`Server.java`

```
public class Server implements TestConstants {

   public boolean test(int j) {
         return j == MyClass.j;
   }
}
```

The value of the expression "`j == MyClass.j`" depends on whether the client and the server reside in one address space or not. If the application is configured that the client and server components run in one Java Virtual Machine, the expression will be always true. If the client and the server are separated the expression can be also `true`, but that is highly improbable.

All access modifiers are allowed, that is **`public`**, **`protected`**, **`private`** and fields without any modifier. All instance fields will be serialized, it does not depend on their visibility.

Both static and instance methods are allowed as well as nested and inner classes. But static methods in this case are not much usual because the non-final static fields should not be referenced. Thus in the body of static methods are available only final static fields, static methods and nested classes and that can be mostly replaced by some expression.

If an object contains some subobjects they are serialized and transferred to the server too (and thus they have to be also serializable). All objects accessible from to "root" object form an oriented graph. Objects represent nodes of the graph and references between them oriented edges. When the object is serialized (and recursively all its subobjects) and passed to the server, the topology of the graph is preserved (that means the relations between objects are the same as on the client side).

- **Interfaces**

```
interface MyIFaceBase {
   long doSomething(int i);
}

interface Foo {}

public interface MyIFace extends MyIFaceBase, Foo {
   // Constant declarations
   int i = 10;                          // OK
   int j = (int)(Math.random()*100); // Avoid this!

   // Method signatures
   String doSomethingElse(int i) throws MyExcetpion;
}
```

  Usage of interfaces in method signatures is fully supported. The above piece of code shows the definition of `MyIFace`. As it is seen the interfaces besides the method declarations can extends other interfaces and also contain definition of constants. Methods are allowed to throw any exceptions.

  A function can have in its signature an interface. But when the function is called there must be passed some object that implements that interface.

  If an object (that implements the appropriate interface) is passed to a remote method it must be also serializable (implement `java.io.Serializable`). If this rule is broken a runtime exception will be thrown.

  But be careful with constants (they are implicitly `public`, `static` and `final` and that cannot be changed). Usage of non-constant initializers (see the `(int)(Math.random()*100)` expression) is not recommended. It is the same problem as with static final fields in objects (see the previous section).

- **Business Interfaces**

  To a definition of a business interface apply the same rules as to any other interface. There can be used inheritance, constants declarations as well as exceptions in functions. But it is strongly recommended to avoid using non-constant initializers in constant declarations.

  As has been said even if a signature of some function contains some interfaces when the function is called the actual parameters are objects (implementing that interfaces).

  There is a difference between passing “ordinary” objects and the “remote” ones but the user usually do not have to concern with it:

  - If a user defines a local object implementing some business interface it behaves like any other object (see previous section discussing about objects). When it is passed to the server its internal state is serialized in order to be transferable over the network and on the other side there is created an “image” from it, new identical copy. Thus the type of the object and values of its fields correspond on both sides. Here is an example:

```
MyLine line =
  new MyLine(0.0, 0.0, 1.0, 1.0);

// Get the remote interface
Test remote = . ..;

System.out.println(remote.test(line));
```

Client

```
public double
compute(MyLine line) {
   return line.getLength();
}
```

Server

The type `MyLine` can implement some business interface, for example `GraphicsObject`. But that fact does not denote that it is a remote object. In the example the object `line` (of type `MyLine`) is a “normal” local object.

The client creates an object `line` which constructor has four parameters (coordinates of the endpoints (x1, y1, x2, y2)). Then it gets a remote reference and call a remote method which is implemented on the server. It has one parameter of the type `MyLine`. The client object created using the constructor `MyLine(0.0, 0.0, 1.0, 1.0)` is passed to the server (the coordinates are copied) and on the server side the length is computed. The result is returned back to the client and printed.

- The remote objects (alias remote references) are not serialized. Instead of that they are replaced by a wrapper containing some information describing that object. Server then creates from it a new connector unit. It do not have to be of the same type, important thing is that it implements the same interface and all calls forwarded through it are processed by the same implementation (some server) so it has an identical behavior. It is showed in the following example:

```
// Get the remote objects
Test_1 r_1 = ...;
Test_2 r_2 = ...;

System.out.println(
    r_2.getClass());
);
System.out.println(
    r_1.test(r_2)
);
System.out.println(
    r_2.message();
}
```

Client.class

JVM1

```
public String
test(Test_2 r_2) {
  System.out.println(r_2.getClass());
  System.out.println(r_2.message());

  return
  r_2.getClass().toString() + ", "
    + r_2.message();
}
```

Server_1.class (implements Test_1)

```
public String
message() {
   return "Hello";
}
```

JVM2

Server_2.class (implements Test_2)

```
Output:
class generated.A0009.RMIStub
class Server_2, Hello
Hello
```

```
Output:
class Server_2
Hello
```

The client holds two remote references. One pointing to `Server_1` (in the variable `r_1`) and the other to `Server_2` (in the variable `r_2`). Then it prints the class of the object contained in the variable `r_2`. Because it is a remote reference, it is represented e.g. by the proxy `generated.A0009.RMIStub`. The next call is through `r_1`. It has one argument where `r_2` is passed. `Server_1` then prints the class of that argument. Although on the client side the `r_2` variable is pointing to `generated.A009.RMIStub`, on the server side the argument `r_2` points directly to `Server_2` because `Server_1` and `Server_2` are in the same address space, thus no mediator is needed. That is why the expression `r_2.getClass()` returns different values on the client and on the server. But it does not change the fact that on both sides it implements the same interface and the `message()` method will return the same string "Hello".

Chapter 7

Implementing support for CORBA

7.1 Overview

CORBA has many implementations and bindings into various programming languages. Concretely with Java language there are associated two technologies, Java IDL [36] and Java RMI over IIOP [37].

Java IDL is a “direct” implementation of CORBA, it is built on the Interface Definition Language providing standards-based interoperability and connectivity.

Java RMI-IIOP is more "Java oriented" but it is still possible to interoperate with any CORBA-compliant language. The main benefit for the Java language is that there is no need to use IDL, it is possible to write a whole application in Java using the RMI API.

Both technologies have its “pros and cons”. Here is a description of that technologies to better understand their differences and also benefits and restrictions which they bring.

7.1.1 Java IDL

Java IDL is a technology for distributed objects. It is based on IDL (Interface Definition Language) which is language neutral and thus it allows to interconnect programs written in different programming languages.

One of the main parts of the CORBA technology is IDL. It is created at time of designing an application. A programmer decides which components the application constitutes of and how they interact with each other. IDL defines which interfaces will be used and thus it forms a headstone of the application. Here is a sample `Hello.idl` which will be used in consequent examples:

```
module HelloWorld {
    interface Hello {
        string message();
    };
};
```

It is very simple but satisfactory for showing the development process. `module` is a group of related stuff (interfaces, types, ...). The Java `package` is very close it and as it will be seen later it is indeed generated from it. The body of `module` in the example consists only of one `interface`. It is direct equivalent for Java `interface`, it states which methods some object provides.

When the IDL interfaces has already been designed it is time to generate bindings for Java from them. For that purpose is supposed the tool `idlj`. It implicitly generates only client-side bindings, for generation also for the server the `-fall` option must be used:

```
idlj -fall Hello.idl
```

The command above generates six files and store them into the `HelloWorld` directory (because of the name of the module). Here is a list of that files and a brief description what they are used for:

- HelloOperations.java – contains all methods from the `Hello` interface, in this case only the `message()` function:

  ```
  package HelloWorld;

  public interface HelloOperations
  {
    String message ();
  } // interface HelloOperations
  ```

- Hello.java – is empty interface but it extends `HelloOperations` which contains all methods from the IDL interface. It also extends `org.omg.CORBA.Object` providing standard CORBA object functionality. It is used in the server and the client for holding references to CORBA objects.

  ```
  package HelloWorld;

  public interface Hello extends HelloOperations,
  org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity
  {
  } // interface Hello
  ```

- HelloPOA.java – an abstract class that is used on the server side. It extends `org.omg.PortableServer.Servant` and implements `HelloOperations`. The servant then extend this class and is supposed to implement all functions from the `HelloOperations` interface.
- _HelloStub.java – is a client's stub. It provides low level functionality (marshaling arguments, communication with the server...) and implements `Hello` interface. Client's remote references are in fact references to this class.
- HelloHelper.java – contains auxiliary functions, in particular the method `narrow()` for casting from the general CORBA object (`org.omg .CORBA.Object`) to the more specific `Hello` interface. It is used both by the server and the client.
- HelloHolder.java – a class that contains public field of the type `Hello`. It is used for supporting `inout` and `out` IDL parameters (they are intended to

contain return values of functions). Because the Java language do not allow return values in parameters, output values of the type `Hello` are wrapped into the `HelloHolder` object where it is possible to pass them.

When the server is implementing some interface from IDL it must provide definition for all its methods. In *Inheritance Model*[1] servants (for type `Hello`) have to extend from `HelloPOA` which provides POA (Portable Object Adapter) functionality.

Here is an example of the servant:

```
class HelloServant extends HelloPOA {

  public String message() {
    return "Hello world!\n";
  }
}
```

The server must create an instance of the `HelloServant` and initialize the CORBA subsystem. It is not important how exactly it is achieved, it is enough to know that generated classes (using `idlj`, see previous section) are used and the clients have facilities for obtaining the reference to the remote object.

A client has to also initialize the CORBA subsystem. Then it gets the remote reference and cast it to the `Hello` interface (using the `narrow()` method of the `HelloHelper` class). Afterwards it can use the remote object like as it was a local one.

Client has a number of ways how to get a reference to a CORBA object, e.g. through the COS Naming Service [38] or using a stringified object reference [39]. These facilities are in general independent on the background CORBA technology actually being used and thus they are useful with the Java RMI over IIOP as well. We have decided for the stringified object reference because there is no need for running additional naming service.

7.1.2 Java RMI over IIOP

Java Remote Method Invocation over Internet Inter-ORB Protocol technology [40] is part of Java 2 Platform Standard Edition (J2SE [41]). It allows to use the Java RMI within the CORBA technology. If the whole application is written in the Java programming language the “ordinary” Java RMI (over JRMP protocol) is probably more suitable. But when it is also intended to cooperate with other CORBA objects (possibly written in different language) here it comes in handy.

This technology is more natural (than Java IDL) for Java programmers because they can design remote interfaces directly in Java (using RMI). In contract to Java IDL where an IDL file is first created and then a Java remote interface using `idlj` is generated. It is also interesting for programmers using Enterprise Java Beans because that technology is also RMI-based.

1 There also exist *Tie Model*. For the differences see idlj manual pages.

The following text describes a development process of an application based on the Java RMI over IIOP (using POA-based server-side model).

At first a programmer has to design a RMI remote interface. The interface must satisfy some requirements in order it can be used in the remote communication (see Chapter 6.1). Here is an example interface which will be used next through this section. In contains only one function returning string:

```
public interface Hello extends java.rmi.Remote {
  public String message() throws java.rmi.RemoteException;
}
```

A server object then have to implement the interface so clients can use it. A class that contains the implementation must extend `javax.rmi.PortableRemoteObject` as it is shown in the example below:

```
import java.rmi.RemoteException;
import javax.rmi.PortableRemoteObject;

public class HelloServant extends PortableRemoteObject
 implements Hello {

  public HelloServant() throws RemoteException {
    // Empty
  }

  public String message() throws RemoteException {
    return "Hello World!";
  }
}
```

Because of extending `PortableRemoteObject` the `HelloServant` class can be used as a remote object. It is important that constructor declares to throw `java.rmi.RemoteException` otherwise the java compiler will report an error:

```
HelloServant.java:4: unreported exception java.rmi.RemoteException;
must be caught or declared to be thrown.

      public class HelloServant extends PortableRemoteObject ...
             ^
1 error
```

The server has to contain some code that creates the `HelloServant` object and exposes it to the clients (e.g. via *COS Naming Service* or using stringified object reference). That code may be placed directly in `HelloServant` itself or in a completely separated class.

In order that the server ant the clients can communicate there must be in addition created stub and skeleton files. They are generated from the implementation class (`HelloServant` in this case) using the `rmic` utility. `HelloServant` must be first compiled because `rmic` operates directly on a binary code:

```
javac -classpath . HelloServant.java
```

The `javac` command generates the file `HelloServant.class`. Then it can be processed by the `rmic` utility with the `-poa -iiop` options (`-iiop` causes `rmic` to generate IIOP stub and tie classes rather than JRMP stub and skeleton classes; `-poa` changes the inheritance from `org.omg.CORBA_2_3.portable.ObjectImpl` to `org.omg.PortableServer.Servant`, see `rmic` manual pages). The command `rmic -poa -iiop` takes one or more qualified class names. It introspect implementation classes and generates the necessary files – for each remote object implementation class creates a tie (skeleton) and a stub.

The following command

```
rmic -poa -iiop HelloServant
```

creates these files:

- `_Hello_Stub.class` – a client's stub generated for the `Hello` interface. If the `HelloServant` class was implementing another remote interface an another stub would be also generated.
- `_HelloServant_Tie.class` – a server's skeleton.

As mentioned before the client has to somehow obtain a reference to the remote object. We have decided to use stringified object reference because the server can pass the remote address to the client in form of string and there is no need for running any extra naming service (for example `orbd`).

The class `org.omb.CORBA.ORB` offers the method `object_to_string()` which converts an instance of type `org.omg.CORBA.Object` to its address in the distributed environment. That address is quite a long string and has this form:

```
  IOR:0000000000000033524d493a67656e6572617465642e4130303030303030362e496
e746572666163653a303030303030303030303030303030300000000000010000000001000
0002000000000000100010000000205010001000100200001010900000001000101000000
0026000000020002
```

The client then uses the following sequence of code to obtain the remote reference from the variable `stringifiedRef` of the type `java.lang.String` holding the address:

```
org.omg.CORBA.Object corbaObj =
    orb.string_to_object(stringifiedRef);

Hello target = (Hello) javax.rmi.PortableRemoteObject.narrow(
    corbaObj, Hello.class);
```

The function `string_to_object()` returns the common CORBA type `org.omg.CORBA.Object`. It must be narrowed to a more specific type in order to be useful. The static method `javax.rmi.PortableRemoteObject.narrow()` takes two parameters, the first is a CORBA object to be narrowed and the second an instance of `java.lang.Class` representing a target type, in this case `Hello.class`. The result must be furthermore casted to the remote interface type because the `narrow()` function returns only the general Java type `java.lang.Object`.

Although RMI-IIOP is a powerful technology it has some restrictions too. Here is a list taken from "RMI-IIOP Programmer's Guide" [42]:

- *Make sure all constant definitions in remote interfaces are of primitive types or String and evaluated at compile time.*
- *Don't use Java names that conflict with IDL mangled names generated by the Java to IDL mapping rules. See section 28.3.2 of the Java Language to IDL Mapping specification [43] for the Java to IDL name mapping rules.*
- *Don't inherit the same method name into a remote interface more than once from different base remote interfaces.*
- *Be careful when using names that differ only in case. The use of a type name and a variable of that type whose name differs from the type name only in case is supported. Most other combinations of names that differ only in case are not supported.*
- *Don't depend on runtime sharing of object references to be preserved exactly when transmitting object references across IIOP. Runtime sharing of other objects is preserved correctly.*
- *Don't use the following features of RMI:*
    - *RMISocketFactory*
    - *UnicastRemoteObject*
    - *Unreferenced*
    - *The Distributed Garbage Collection (DGC) interfaces*

7.2 Comparison of the Java IDL vs. RMI-IIOP

There is a big difference in development process whether it is started with an IDL file or with a Java interface. Sometimes a programmer can choose which one he starts with, especially when writing a whole application from scratch. But it is not always possible, it can be forced by a part of an application already written. The following subsections discuss the two cases.

7.2.1 Starting from an IDL

There can be more reasons why start with an IDL file. The first is simply that it already exists, e.g. the application is already designed, some components are done and some are required to be completed, where IDL prescribes their API. The other reason may be to take into account future possibility of usage in a heterogeneous environment and make the application ready for it. In such a case it is much more reasonable to start with the IDL file and then from it generate Java interfaces than start with Java and to that write a corresponding IDL. The Java language is much more specific than IDL and it can be very complex to generate an appropriate IDL file from a Java interface than vice versa.

Here are described the two technologies, their usage and consequences when a programmer starts with IDL:

- **Java IDL**

  It very natural to use this technology when a programmer starts with IDL. Java IDL is directly designed for it, for the development process see Chapter 7.1.1.

- **RMI-IIOP**

  For RMI-IIOP it is necessary to generate a compiled RMI interface which would correspond to the given IDL file. It is possible in number of ways, here are some potentially solutions:

  - The `idlj` utility besides the number of generated files creates also `HelloOperations.java` which contains the corresponding Java interface (in form of a source code) to the IDL interface. But RMI-IIOP requires binary code and in addition it must satisfy some requirements to be used with RMI (see Chapter 6.1). Thus the `HelloOperations.java` file has to be adjusted. First of all it has to extend `java.rmi.Remote` and each method must throw `java.rmi.RemoteException`). It can be achieved by the already existent type operators `rmi(java_interface('HelloOperations'))`. `java_interface()` takes as an argument Java type (compiled), thus `HelloOperations.java` must be first compiled into `HelloOperations.class` before use. From the created type `rmi(...)` is then generated Java source and compiled into a Java class file. The whole process is depicted in Figure 7.1.

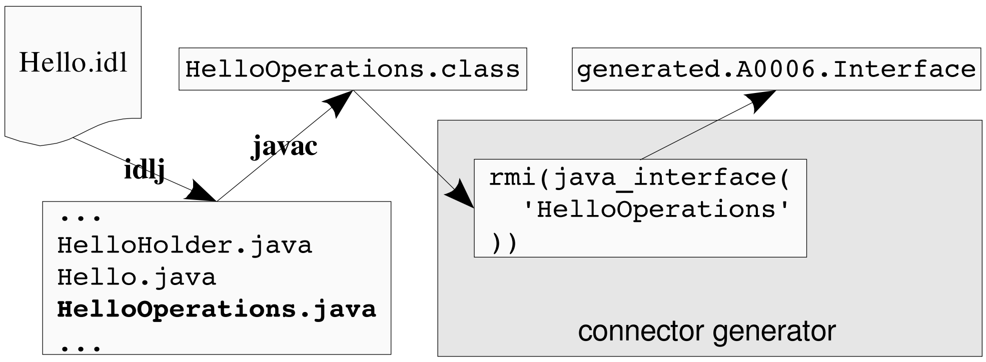


Figure 7.1: A possible solution of creating of a RMI interface from IDL

  But an IDL interface can be much more complex than `Hello` from the `Hello.idl` file and use other types defined by IDL. Moreover the IDL file itself can include other IDL files. Thus a generation of some Java interface can result in creation of many types on which the interface is dependent. Here is an example of a little more complex IDL file:

```
module HelloWorld {
      struct Foo {
            string symbol;
```

```
                long at_time;
                double price;
                long volume;
        };
        exception SomeException {};

        interface MoreCompexHello {
                string message(out Foo foo)
                        raises(SomeException);
        };
};
```

... and its corresponding `MoreCompexHelloOperations` interface in Java (the struct which the interface depends on is not shown):

```
package HelloWorld;

public interface MoreCompexHelloOperations
{
        String message (HelloWorld.FooHolder foo) throws
                HelloWorld.SomeException;
} // interface MoreComplexHelloOperations
```

- The other solution is to process the IDL file directly without any intermediate files (like `HelloOperations.java` or `HelloOperations.class`) as in the previous method. In such a case the IDL file have to be parsed into internal data structures. Fortunately there is already a parser which is used also by the `idlj` utility. The class `com.sun.tools.corba.se.idl.toJavaPortable.Compile` contains many useful methods that the connector generator can use. To the connector type system can be added some type operators like `idl_interface()` and `to_java_interface()`. A resulting type expression which is creating the RMI interface could for example look like this: `rmi(to_java_interface(idl_interface('Hello.idl', 'HelloWorl','Hello')))`. Since the IDL file can include more modules and each module can contain many interfaces, the operator `idl_interface()` must specify which interface from which module it is supposed to represent.

  The type operator `to_java_interface()` maps an IDL interface to Java. It is reasonable to do the mapping on a base of some standard. The document 'Java IDL: IDL to Java Language Mapping' [44] shows the correspondence between OMG IDL types and Java types. For the full OMG specification see 'Java to IDL Language Mapping, v1.3' [45].

  But the generation of the Java interface do not have to be easy. The IDL interface can be using other types defined in the IDL file and even in a completely separated file (included via the `#include` construct).

  Moreover when the necessary class files are created at the connector generation the programmer cannot use them earlier when designing and developing the application. Many IDEs (Integrated Development

Environments) are capable of immediate syntax checking and code completion. It is with absence of generated sources impossible and the environment will report an error that it does not know the type (which is generated later).

It seems to be more reasonable to use the `idlj` utility within the development process. A programmer should generate necessary sources manually into some folder and then set up that folder in the IDE's build path. Thus the development environment will be aware of the generated types.

7.2.2 Starting from a Java Interface

This will be probably much more frequently case than designing and developing an applications from IDL because users mostly do not need such a universality that the IDL brings. Here is again an overview how the two technologies (Java IDL vs. RMI-IIOP) are used when starting with some Java interface:

- **Java IDL**

  We highly discourage to use Java IDL with interfaces originally designed in Java because Java is a much more specific and complex language than IDL. A lot of mappings from Java to IDL are problematic and brings only problems, it is usable only with very simple interfaces.

  To generate an IDL file for from a Java interface the utility `rmic` with the option `-idl` can be used. The arguments of `rmic` should be full qualified java types.

  Here is a list of some examples which cause serious problems even if they are very simple:

  - Hierarchy of Java interfaces in some ordering can even into lexically incorrect IDL files (generated via `rmic`)! For example here is a definition of two Java interfaces:

    ```
    public interface A {}
    ```

    and

    ```
    public interface B extends A, java.rmi.Remote {
    }
    ```

    A simplified output of the '`rmic -idl B`' command is:

    ```
    #include "A.idl"

    interface B:  {
    };
    ```

    There is an error, after the colon an interface name is expected (which the B class inherits from). Here is an error message of `idlj` running on `B.idl`:

```
> idlj B.idl
B.idl (line 15):  Expected `<identifier>'; encountered `{'.
      interface B:  {
```

  - An occurrence even of a very simple Java exception causes the generation of many IDL files because IDL exceptions must reflect a hierarchy of Java exceptions. Running `rmic -idl` on the interface below results in the generation of 24 IDLs.

    ```
    public interface A {
          public void doSomething() throws Exception;
    }
    ```

    Creating stubs and skeletons from that IDLs (using the `idlj` compiler) leads into yet more Java sources which some of them cannot be even compiled (because the `idlj` compiler produces incorrect code).

  Thus because of former reasons the Java IDL technology seems to be useless when starting development of an application using Java interfaces.

- **RMI-IIOP**

  RMI-IIOP is straight designed to be used within Java interfaces. They just have to be adjusted to be remote. For this purpose is the type operator `rmi()` perfectly suitable.

7.3 Implementation details

We have implemented Java RMI over IIOP because it is possible to use it with the native Java interfaces as well as with IDL.

The implementation of support for CORBA in the connector is very similar to integration of RMI described in Chapter 6.3. In fact CORBA is also some kind of Remote Method Invocation, it is based on the calling of remote methods.

Because configuration files and the whole concept are very similar to Java RMI there will be in the next sub-chapters described only significant differences. The `RMIObjectAdapter` class is used for handling of remote references because the structure of stub/skeleton model remains the same. Instead of `RMIStub.class` there is `CorbaStub.class` and `RMISkeleton.class` is replaced by `CorblaSkeleton.class`. Similarly `corba_stub.xml` is a configuration file for the `CorbaStub` class and analogously `corba_skeleton.xml` for the `CorbaSkeleton` class.

7.3.1 Configuration of CORBA stubs and skeletons

As was described in Chapter 5.3 element configuration files contain besides other stuff also virtual “cost” of a particular architecture, signatures of ports, remote reference names and build scripts. In a case of CORBA stub the build scripts are very similar to those in Java RMI, there are no special compiler options. The main differences are in this part of the `corba_stub.xml` file:

```
<architecture cost="7">
  <port name="call" signature="I"/>
  <port name="line">
    <signature-entry ref-name="corba" type="client" signature="rmi(I)"/>
  </port>
</architecture>
```

The cost of the architecture is higher. Java RMI has number 5 whereas here is 7. It should express that usage of CORBA middleware is a little more demanding on systems resources. But it is not so significant, it depends on a programmer's personal opinion what he thinks, it need not reflect the reality. But based on that value the generator is trying to choose the “best possible” configuration so it influences the resulting efficiency of the connector.

The other difference is that it is using the “corba” reference name instead of “rmi”. There could be any string that differs from the other reference names.

The file `corba_skeleton.xml` is very similar but it has in addition two options `-poa` and `-iiop` in the `rmic` compiler specification to create the CORBA stub and Tie (skeleton).

```
<command action="rmic">
    <param name="class" value="CorbaSkeleton"/>
   <param name="add_param" value="-poa"/>
   <param name="add_param" value="-iiop"/>
</command>
```

7.3.2 `CorbaStub.class`

The `CorbaStub` class is analogous to `RmiStub` marked with the proper annotation so `RMIEncoder` can handle it correctly (see Chapter 6.4.1).

```
@org.objectweb.dsrg.connector.rmi.ConnectorInterface(
      "generated.A0000000A.Wrapper")
public final class CorbaStub
      implements
      ...
```

The wrapper is generated exactly in the same way as within `RMIStub`.

A constructor of `CorbaStub` besides an other initialization sets up also a protected field `orb`.

```
    protected ORB orb;

    // Constructor
    public CorbaStub (
            org.objectweb.dsrg.connector.ConnectorUnit parentUnit,
            boolean isTopLevel)
        throws org.objectweb.dsrg.connector.ElementLinkException {
        ...
        // create and initialize the ORB
        orb = ORB.init(...);
        ...
```

The field orb is then used in the client's function bindElRemotePort() to obtain a remote reference (see Chapter 5.4):

```
public void bindElRemotePort (
      String portName,
      org.objectweb.dsrg.connector.RemoteRefBundle refBundle)
throws org.objectweb.dsrg.connector.ElementLinkException {
      ...
      refBundle.getRef("corba");
      org.omg.CORBA.Object obj = orb.string_to_object(
            corbaRef.stringifiedRef);

      target = (generated.A00000006.Interface)
            PortableRemoteObject.narrow(
                  obj, generated.A00000006.Interface.class
            );
```

The variable corbaRef.stringifiedRef contains stringified object reference (of the remote object). The statement orb.string_to_object(corbaRef .stringifiedRef) returns a CORBA object which the stringifiedRef points to. It must be then narrowed and casted to an appropriate type.

The remain of the class is the same as in the RMIStub class. For each method from the business interface is generated a corresponding method in CortbaStub (see Figure 6.32). The are forwarding method callings from the client component to the server. If some arguments are of an object type the RMIObjectEncoder() method is also used to ensure a proper behavior with remote references.

7.3.3 CorbaSkeleton.class

The CorbaSkeleton class acts as a CORBA servant and thus it must inherit from javax.rmi.PortableRemoteObject which makes the object usable with the CORBA engine:

```
public final class CorbaSkeleton
extends
      javax.rmi.PortableRemoteObject
implements
      org.objectweb.dsrg.connector.ElementRemoteServer,
      org.objectweb.dsrg.connector.ElementLocalClient,
      ...
```

A constructor initializes ORB and gets a *tie* for itself. The tie is something like skeleton, it provides a lowlevel functionality. When the tie is obtained it must be registered within CORBA via the function activate_object_with_id(). It makes the CorbaSkeleton object ready for use. Finally its stringified reference is stored to the ior field.

```
protected String ior;

// Constructor
public CorbaSkeleton (
      org.objectweb.dsrg.connector.ConnectorUnit parentUnit,
      boolean isTopLevel)
      ...
{
      ...
      ORB orb = ORB.init(new String[] {}, p);
      ...
      Servant tie = (Servant)Util.getTie( this );
      tPOA.activate_object_with_id( id, tie );
      ior = orb.object_to_string(tPOA.create_reference_with_id(id,
            tie._all_interfaces(tPOA,id)[0]));
      ...
```

A function `lookupElRemotePort()` adds the stringified reference stored in the `ior` field to the global “reference bundle” (see Chapter 5.4) through the method `addRef()`. Thus the client can retrieve it through the opposite function `refBundle.getRef()`.

```
public org.objectweb.dsrg.connector.RemoteRefBundle lookupElRemotePort (
    String portName)
throws org.objectweb.dsrg.connector.ElementLinkException {
    ...
    if ("line".equals (portName)) {
        result.addRef (
            new org.objectweb.dsrg.connector.RemoteRef("corba", ior)
        );
    ...
```

The remain of the class is the same as in the `RMISkeleton` class. For each method from the business interface there must be corresponding method in the `CorbaSkeleton` which forwards callings to the server component. Object arguments must be first processed with the `RMIObjectDecoder` method (see Chapter 6.4.3).

7.4 Summary

We have integrated the RMI-IIOP technology into the connector generator. It brings the possibility to use the connector with CORBA compliant components.

When at a time of writing distributed application is in advance taken into account that more programming languages can be used, we highly recommend to start with IDL. Starting with some language specific interface and then retroactively designing IDL leads mostly into difficulties. In this chapter we have discussed differences between using the two technologies (Java IDL and RMI-IIOP).

The following table shows which interfaces are usable with which technologies. It is seen that RMI-IIOP can be used both with Java interfaces and IDL, but that IDL must be first converted into Java using `idlj` utility.

	Java IDL	RMI-IIOP
IDL	✓	✓ (using `idlj` to convert IDL to a Java interface)
Java Interface	✗	✓

Usability of different types of interfaces (IDL vs. Java) with the distinct technologies (Java IDL vs. RMI-IIOP)

The connector generator can be also utilizing both types (Java IDL and RMI-IIOP) and at a time of generating of a specific connector use the more appropriate one. It is probably the best solution.

Chapter 8

JMS

The *Java Message Service* (JMS) [26][46] API is a message oriented API which allows sending messages between two or more participants. It supports both kinds of messaging – point-to-point as well as publish/subscribe schemes.

But the JMS API is only an "interface", there can be various middlewares that implements that API.

Java Platform, Enterprise Edition (Java EE) implements the JMS API, but it is "huge" and unnecessary complex for the usage in the connector generator.

There was a requirement to find a middleware that would be easy to use but still providing enough functionality.

It seems that *Apache ActiveMQ* [27] is a suitable for this task. It is popular and powerful open source message broker[1] which implements the Java Message Service (JMS). Apache ActiveMQ supports many enterprise features, cross language clients (C, C++, C#, Java, Ruby, PHP, Perl, Python) and protocols.

8.1 The JMS API Overview

The JMS API consists of several interfaces. Figure 8.1 shows the most important of them and their relations.

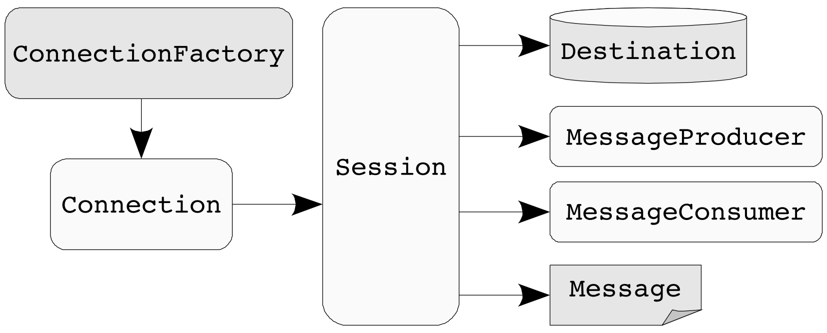


Figure 8.1: Relations between JMS interfaces. An arrow means that one object is creating the other.

1 Message broker is a program that translates messages from a messaging protocol of the sender to a messaging protocol of the receiver.

ConnectionFactory is used to create a connection to a JMS server. It hides a lowlevel configuration like an URL of the JMS server. ActiveMQ provides ActiveMQConnectionFactory which do not implement the ConnectionFactory interface but is also used for creation of javax.jms .Connection. A constructor can have a user's name, password and URL of the JMS server as paramaters.

```
ActiveMQConnectionFactory connectionFactory = new
      ActiveMQConnectionFactory(userName, password, brokerURL);
```

The connection can be created from the connection factory. The connection represents a binding between the client and the JMS server and allows to create a session.

```
// Create the connection
Connection connection = connectionFactory.createConnection();
connection.start();
...
connection.close();
```

Before the application can receive messages it must call the connection's method start(). And when it is not more needed the method close() should be called to release resources.

A session represents a single-threaded messaging context. It means that all messages in one session will be “serialized” and received one by one. The session also supports transactions.

```
// Create the session
Session session = connection.createSession(false,
      Session.AUTO_ACKNOWLEDGE);
```

The first parameter in the example means that the session should not use transactions and Session.AUTO_ACKNOWLEDGE ensures that messages will be automatically acknowledged when they will be received.

A destination is an object that represents a location where messages are delivered to and received from. It can be either a queue or a topic.

```
// Create the destination
Destination destination = session.createTopic("some_topic_name");
```

A producer is used for sending messages to a destination. It can be created by session's createConsumer(Destination dest) function with one argument that denotes a destination. A user can create also a generic sender that specifies the destination at the time the message is sent.

```
// Create the producer
producer = session.createProducer(destination);
```

A message interface is extremely flexible and provides numerous ways to customize the contents of a message. In the connector generator for the testing purposes it is enough to use only textual messages. The message can be then sent using the producer.

```
// Create some message
TextMessage message = session.createTextMessage("some text");

// Sent the message
producer.send(message);
```

A message consumer receives messages from some destination. It can receive them synchronously or asynchronously from both queues and topics. It is also created by the session.

An object which is supposed to receive messages asynchronously must implement the `MessageListener` interface, which contains only one method `onMessage()`. That object is then passed to the method `setMessageListener()` of the consumer.

```
// Create the consumer
consumer = session.createConsumer(destination);

// Listener define the onMessage() method
MessageListener listener = ...;

// Set the handler for asynchronous messages
consumer.setMessageListener(listener);
```

8.2 Implementation details

Messaging can be used in many ways. We have implemented the publish/subscribe messaging, where clients send messages through *topics*, and asynchronous mode, thus clients do not have to poll server for new messages and instead they are given a message when it arrives in the topic.

8.2.1 Connector architecture

The connector architecture for the messaging communication style uses the client/server model. It consists of many client units and one server which acts as a control unit. Client units are then attached to the client components which can send and receive messages. The server unit is standalone with no component binded, it is supposed to operate as a JMS server. Figure 8.2 shows the configuration file for the messaging communication style.

```
<connector name="messaging">

  <architecture>
    <unit name="client_unit" type="messaging_client_unit"
        cardinality="multiple"/>

    <unit name="server_unit" type="messaging_server_unit"
        cardinality="one"/>

    <binding>
      <port element="client_unit" port="line"/>
      <port element="server_unit" port="line"/>
    </binding>
  </architecture>

</connector>
```

Figure 8.2: org/objectweb/dsrg/congen/conrep/connectors/messaging.xml

Figure 8.3 then depicts an architecture of the connector with attached client component. The unit `messaging_client_unit` has two local ports, one for sending messages and one for receiving and one remote port `line` for communication with the server. The unit can work dynamically, it can be either a sender, a receiver or both (but acting in both roles is not much useful because it causes that sent messages by the client component are also in turn received by the same component).

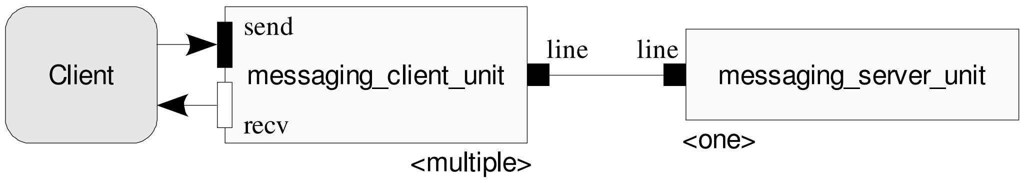


Figure 8.3: Graphical illustration of the connector architecture for messaging with an attached client component

Because of more flexibility `messaging_client_unit` and `messaging_server_unit` can be next subdivided as shown in Figure 8.4.

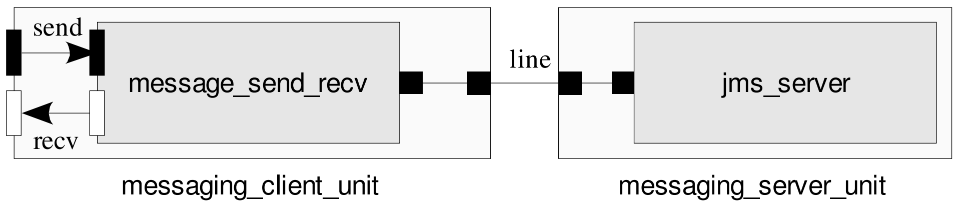


Figure 8.4: Client's and server's unit refinement

If the client component wants to send some message it must bind the `send` port (it must obtain its reference). The port `send` will be always exposed to the client because it is implemented as a Java interface that `message_send_recv` implements and the client component holding a reference to it can any time call a method on it. That is why the message_send_recv element must always be ready to send messages. It send messages through an internal message producer, but message_send_recv can create the message producer dynamically, by the first call on the send port. If the client component never sends a message the internal producer need not be instantiated and that can spare system resources.

On the other hand the port recv is more controllable by the message_send_recv unit. Here is the reference opposite, message_send_recv is holding the reference to the client component. That reference is set up by the client through the function bindElPort("recv", listener) where the object listener must implement the onMessage() method. Until the bindElPort method is called (and thus message_send_recv is holding null pointer to the client component) there is no need to create message_send_recv's internal message receiver.

The method unbindElPort() can be also called. It unbinds the recv port and it can have the semantics that the client do not want to listen to messages anymore and thus the internal message_send_recv's internal message consumer should be closed.

8.2.2 Implementing the jms_server

The jms_server element type (see Figure 8.4) is located on the server side. It is nested inside the messaging_server_unit and should act as a JMS server (see Figure 8.4).

One concrete implementation of the jms_server type is the JMSServer class. Here is its main code taken from the JMSServer's constructor, it creates an embedded JMS server and runs it.

```
import org.apache.activemq.broker.BrokerService;

public final class JMSServer
      implements
            ...
{
      // Constructor
      public %CLASS% (
            org.objectweb.dsrg.connector.ConnectorUnit parentUnit,
            boolean isTopLevel)
            throws ...
      {
            ...
            BrokerService broker = new BrokerService();
            broker.setUseJmx(true);
            broker.addConnector(.../* bind address */);
            broker.start();
      }
}
```

The method broker.addConnector() set the JMS server's bind address (which port and address the server should listen on). That address should be configurable and the client must know it because it connects to that address. The server send round messages between clients that are connected to it.

8.2.3 Implementing the message_send_recv

The client side is more complex. The class `MessageSendRect` implements the `MessageSender` and `MessageListener` interfaces. The interface `MessageSender` contains one method `send(String message)` through which the client component can send messages. The class `MessageSendRecv` implements `MessageListener` because it acts as a proxy. Messages received from the server are first caught by this class and then forwarded to the client.

```
...
import javax.jms.*;

public final class MessageSendRecv
      implements
            ...
            org.objectweb.dsrg.connector.messaging.MessageSender,
            MessageListener {

      public void bindElPort (String portName, Object target) {
                  ...
                  // Listen for messages from the server,
                  // 'this' object acts as a proxy
                  consumer.setMessageListener(this);
                  ...
      }

      public void unbindElPort (String portName) {
                  ...
                  // Stop sending messages
                  consumer.setMessageListener(null);
                  ...
      }

      // Establish a remote binding
      public void bindElRemotePort (..., RemoteRefBundle refBundle) {
                  ...
                  // Create the connection.
                  ActiveMQConnectionFactory connectionFactory =
                        new ActiveMQConnectionFactory(...);

                  connection = connectionFactory.createConnection();
                  connection.start();

                  // Create the session
                  session = connection.createSession(false,
                        Session.AUTO_ACKNOWLEDGE);

                  // Get the address of the destination
                  RemoteRef activeMQRef = refBundle.getRef(“messaging”);
```

```
                    // Create the destination
                    Destination destination =
                          session.createTopic(activeMQRef.stringifiedRef);
                    ...
        }

        // Listen for messages from the server
        // and forward them to the client
        public void onMessage(Message message) {
              ...
              // Extract text from the message
              String text = ((TextMessage)message).getText();

              // ... and forward to the client
              client.onMessage(text);
              ...
        }

        public void send(String message) {
              ...
              // Forward to the server
              producer.send(session.createTextMessage(message));
              ...
        }
}
```

The method `bindElRemotePort()` should not be called by the client. It is invoked automatically by the connector. Its purpose is to establish a remote connection with the server. First it creates the connection factory. The `ActiveMQConnectionFactory(...)` has some argument like an address of the JMS server or a password. The URL should be configurable and the same as the `JMSServer` used by startup. Then the connection, session and destination are initialized. The address of the destination is provided by the server, it is simply some string which denotes the name of the topic (or the queue if the point-to-point was used). The address is obtained from the parameter `refBundle` by `getRef()`.

When the client wants to receive messages it calls the method `bindElPort ("recv", listener)`. That method creates an inner receiver and as a listener sets the class `MessageSendRecv` itself. It is because `MessageSendRecv` works as proxy and forwards incoming messages from the server to the `listener` as specified as the second parameter of the `bindElPort()` function. `listener` must implement the interface `org.objectweb.dsrg.connector.messaging .MessageSender` that define the function void `onMessage(String message)` through which `MessageSendRecv` can send message to the listener.

When the client wants to stop receiving messages it can call `unbindElPort("recv")`. Its sets the listener of the `MessageSendRecv`'s internal message receiver to null, but not destroy the receiver. The client can start receive messages anytime again with `bindElPort()`.

The client component can also send messages. For that purpose is supposed the `send()` function. The `MessageSendRecv` object needs an internal message producer through which it can forward messages to the server. But as was said that producer can be created dynamically by the first call of the `send()` function in order to spare system resources.

8.3 Summary

We have implemented support for the JMS API in the connector generator. Messaging needs a different approach than method invocation communication style. A similarity is that it also uses a client/server model, but in messaging the server unit is not attached to any component, it servers only as an arbiter. Most of the functionality is concentrated in the JMS server itself. We have used Apache ActiveMQ which through its flexibility and robustness could be also used as a bridge between different languages like C++, C#, Python and others.

Chapter 9

Related work

The idea of some unification of middlewares and creation of an abstract layer is presented also in the project Arcademis [47][48]. It aims on object oriented middleware and the main purpose why the Arcademic project has arisen is that current convectional object oriented middlewares are monolithic and inflexible to meet the needs of modern rapidly changing technologies. Nowadays there exist many different devices like cell phones, PDAs, etc., with various demands on quality and characteristics of a connection. Arcacemic tries to be highly reconfigurable and addresses the limitations of current middleware implementations. It consists of a set of abstract classes and interfaces that define the general architecture of middleware systems. Arcademic heavily uses design patterns, for example well-known singletons, strategies, factories, decorators and facades. It is designed to be flexible, so that new transport protocols, connection management policies and authentication algorithms can be easily configured. But Arcademic in only a "template" how some middleware could look like, it does not implement any actual functionality. In order to illustrate how Arcademic can be used the document [48] describes a derived middleware from Arcademic named RME providing a remote method invocation for the J2ME/CLDC platform.

The other project using various middlewares for providing inter-component communication is PadicoTM [49]. It aims on parallel computational infrastructures called the grid. The grid encourages the development of new applications in the field of scientific computing. It is used for example for simulation of complex physical processes. Different components of a scientific application may demand different communication paradigms like RMI (Remote Method Invocation) or MPI (Message Passing Interface). However some existing communication middlewares are unable to take benefits of networking technologies available in the grid. And also some low-level communication layers in grid systems were not designed to be able to share resources with several middlewares. The aim of PadicoTM is to allow several communication middleware and runtime to efficiently share the networking resources and use the most suited communication for a specific application.

The last project we want to mention is The Proteus Multiprotocol Message Library [50]. Similarly as PadicoTM it is tied closely with grid systems. It is specialized exclusively on messaging, Proteus is a library for decoupling clients from the messaging protocols. It can communicate through SOAP, JMS or by locally developed binary protocol. But it is designed generally enough and other protocols can be added dynamically without recompiling or halting applications. Proteus is written in C++ but a Java implementation is also planned.

Chapter 10

Conclusion

The goal of this thesis was to extend existing connector generator [1] and provide support for generating and deploying RMI, CORBA, and JMS-based connectors.

The connector generator originally partly supported RMI. Passing of primitive types was implemented and also remote references were supported as well. But passing of object types was completely missing because of several reasons. At first, it was not known whether it would be possible to bypass the limitation that all objects being used within Java RMI must be serializable. It emerged that users always have to be aware of remoting and marking objects as serializable is a reasonable way how to express that. At second, there was a need to deal with remote references inside complex types. We have implemented a special object `RMIObjectAdapter` which uses the Java Reflection API to adjust that inner references in order to allow object transports across the network. Thus the support for Java RMI is now complete.

There was no support for CORBA at all. At first it was mandatory to choose some concrete implementation of CORBA. We have decided for Java RMI-IIOP because it is possible to use it with native Java interfaces as well as with IDL when necessary. We have implemented the support for CORBA in a way that it can manage similar things as the improved support for RMI, that is passing of primitive types, object types and even remote references.

The third goal was to implement different communication style, concretely the JMS API. We have prepared an infrastructure in the connector generator, which comprises configuration files for element types, elements, messaging architecture and templates. With that low-level "background" it was possible to delegate most of the JMS logic to the server side, where we have employed the Apache ActiveMQ as a JMS provider. We have implemented and tested the publish/subscribe model with asynchronous delivery of messages. Concerning a future work it is possible to extend it also by point-to-point messaging, to use more destinations, to add a logger to the architecture, to add support for transactions, to encrypt messages and many others.

Bibliography

[1] Bures, T.: Generating Connectors for Homogenous and Heterogenous deployment, Ph.D. Thesis, advisor: Frantisek Plasil, Sep 2006

[2] Component-based software engineering, http://en.wikipedia.org/wiki/Component-based_software_engineering

[3] Szyperski, C.: Component Software: Beyond Object-Oriented Programming – 2nd edition, 2002, ISBN 0-201-74572-0

[4] Messerschmitt, D. and Szyperski, C.: Software Ecosystem: Understanding An Indispensable Technology and Industry, 2003, ISBN 0-262-13432-2

[5] Software componentry, http://en.wikipedia.org/wiki/Software_componentry

[6] Microsoft corporation, Component Object Model Technologies, http://www.microsoft.com/com/

[7] Sun Microsystems, Enterprise JavaBeans, http://java.sun.com/products/ejb/

[8] OMG, Corba Component Model, http://www.omg.org/technology/documents/formal/components.htm

[9] Common Object Request Broker Architecture, http://en.wikipedia.org/wiki/CORBA

[10] The Object Management Group, http://www.omg.org/

[11] SOFA Component System, http://sofa.objectweb.org/

[12] The Fractal Project, http://fractal.objectweb.org/

[13] Bray, M.: Middleware, 1997, http://www.sei.cmu.edu/str/descriptions/middleware.html

[14] Java remote method invocation, http://en.wikipedia.org/wiki/Java_RMI

[15] Java Virtual Machine, http://en.wikipedia.org/wiki/Java_Virtual_Machine

[16] Serialization, http://en.wikipedia.org/wiki/Serialization

[17] Java Remote Method Protocol, http://en.wikipedia.org/wiki/Java_Remote_Method_Protocol

[18] Sun Microsystems, Java RMI over IIOP, http://java.sun.com/products/rmi-iiop/

[19] Object Management Group, http://www.omg.org/

[20] LipeRMI, http://lipermi.sourceforge.net/

[21] cajo, https://cajo.dev.java.net/

[22] CORBA, http://www.corba.org/

[23] Servant (CORBA), http://en.wikipedia.org/wiki/Servant_(CORBA)

[24] Object Management Groupt, ORB Basics,

http://www.omg.org/gettingstarted/orb_basics.htm

[25] Object Management Group, CORBA/IIOP Specification, http://www.omg.org/technology/documents/formal/corba_iiop.htm

[26] Sun Microsystems, Java Message Service, http://java.sun.com/products/jms/

[27] Apache ActiveMQ, http://activemq.apache.org/

[28] OpenJMS, http://openjms.sourceforge.net/

[29] Bures, T.: A Connector Model Suitable for Automatic Generation of Connectors, Ph.D. Thesis, advisor: Frantisek Plasil, Sep 2006

[30] Prolog, http://en.wikipedia.org/wiki/Prolog

[31] XML-RPC, http://www.xmlrpc.com/

[32] XML, http://www.w3.org/XML/

[33] Sun Microsystems, Java Reflection, http://java.sun.com/docs/books/tutorial/reflect/index.html

[34] Sun Microsystems, Java Annotations, http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[35] Sun Microsystems, Java Collections, http://java.sun.com/docs/books/tutorial/collections/index.html

[36] Sun Microsystems, Java IDL, http://java.sun.com/j2se/1.5.0/docs/guide/idl/index.html

[37] Sun Microsystems, Java RMI over IIOP, http://java.sun.com/j2se/1.5.0/docs/guide/rmi-iiop/index.html

[38] Sun Microsystems, COS Naming Service, http://www-inf.int-evry.fr/COURS/java/jdk1.4docs/guide/idl/jidlNaming.html

[39] Sun Microsystems, Using Stringified Object References, http://java.sun.com/j2se/1.3/docs/guide/idl/tutorial/GSstring.html

[40] Sun Microsystems, Java RMI over IIOP, http://java.sun.com/j2se/1.5.0/docs/guide/rmi-iiop/index.html

[41] Sun Microsystems, Java 2 SE, http://java.sun.com/javase/

[42] Sun Microsystems, RMI-IIOP Programmer's Guide, http://java.sun.com/j2se/1.5.0/docs/guide/rmi-iiop/rmi_iiop_pg.html

[43] OMG, Java Language to IDL Mapping, http://www.omg.org/cgi-bin/doc?formal/01-06-07

[44] Sun Microsystems, IDL to Java Language Mapping, http://java.sun.com/j2se/1.4.2/docs/guide/idl/mapping/jidlMapping.html

[45] OMG, Java to IDL Language Mapping, http://www.omg.org/cgi-bin/doc?formal/03-09-04

[46] Sun Microsystems, The Java Message Service API, http://java.sun.com/javaee/5/docs/tutorial/doc/

[47] Arcedemis, http://www2.dcc.ufmg.br/laboratorios/llp/arcademis/

[48] Arcademis, http://www2.dcc.ufmg.br/laboratorios/llp/publications/Rt2003/LLP002_2003.pdf