Architecture of Distributed Systems

Andreas Rottmann
Department of Computer Science
University of Salzburg
A–5020 Salzburg, Austria
arott@cosy.sbg.ac.at

9. April 2002

Contents

1 Introduction **2**
- 1.1 What is a distributed system? . . . 2
- 1.2 Reasons for Distributing Software . . . 3
- 1.3 Subject of Distribution . . . 3
- 1.4 Goals . . . 3

2 Software Architecture **5**
- 2.1 Architecture Views . . . 5
- 2.2 What is a "good" Architecture? . . . 6
  - 2.2.1 Seperation of Concerns . . . 8

3 Distributed Software Architecture and Software Patterns **9**
- 3.1 Proxy . . . 9
- 3.2 Observer . . . 10
- 3.3 Chain of Responsibilities . . . 10

4 Distribution Models **12**
- 4.1 Client-Server Model . . . 12
- 4.2 Multi-Tier Model . . . 12
- 4.3 Peer-To-Peer Model . . . 13

5 Distributed Software Architecture Concepts **14**
- 5.1 Remote Procedure Calls . . . 14
  - 5.1.1 How an RPC works . . . 14

6 Approaches to Distribution **16**
- 6.1 Distributed Operating System Approach . . . 16
  - 6.1.1 MOSIX . . . 16
- 6.2 Database Approach . . . 17
- 6.3 Programming Language and Middleware-Based Approach . . . 17
  - 6.3.1 CORBA . . . 17
  - 6.3.2 Java RMI . . . 18

A Questions & Answers **19**
- Bibliography . . . 20

Chapter 1

Introduction

As a start, we will try define the term "distributed system" and then elaborate on the reasons for employing distributed systems.

1.1 What is a distributed system?

Various sources give different definitions of the term:

- "A distributed system is a collection of independent computers that appears to its users as a single coherent system." (Tanenbaum)
- "A distributed system is one in which components communicate and coordinate their actions by passing messages." (Coulouris). This implies:
  - Concurrency of components
  - Lack of a global clock
  - Independent failures
- "A distributed system consists of a collection of autonomous computers linked to a computer network and equipped with distributed system software."
- "Distributed systems is a term used to define a wide range of computer systems from a weakly-coupled system such as wide area networks, to very strongly coupled systems such as multiprocessor systems."

In this paper, we will use the following definition:

- A distributed system is a set of physically separate processors connected by one or more communication links.

Each (distributed) system according to the above definition falls in one of the following categories:

tightly coupled: The processors share clock and memory, run one operating system and communicate frequently. The application of tightly coupled systems is also referred to as *parallel processing*.

loosely coupled: Each processor has its own memory, runs its own instance of an operating system and communicates less frequently. The application of loosely coupled systems is called *distributed computing*.

In this paper the focus is on loosely coupled systems, as commonly those are meant when talking about "distributed systems", and they also are more interesting from the software architecture point of view.

1.2 Reasons for Distributing Software

One key factor for the increased application of distributed systems is the over-exponential growth of the bandwidth of communication networks, the growth in this area is several orders of magnitude greater than the increase of processing power of computers. For example, the speed of the backbones used in the Internet has increased from 56kbps over leased telephone lines at the very beginning (ARPANET) to fiber-based communication operating at Gigabit speeds.
This means that the relative expense of remote realization of a computation or remote storage of data has and will continue to decrease over time, thus the decision for distributed processing becoming more frequent.
At the same time, computers are more and more employed not stand-alone, but as part of a networks, such as the Internet.

1.3 Subject of Distribution

The subject of distribution is usually one of the following:

- Data
- Functions
- Load

While the distribution of data and functions is mostly determined by the requirements of the distributed application, load distribution can be fully under control of the distributed system, as with *distributed operating systems* (see Sect. 6.1).

1.4 Goals

With the distribution of data and functions the following, partly overlapping goals are connected ([9], p. 10):

- Decentralization of data and functions of an overall application
- Cooperation of distributed processing units
- Improvement of locality-properties and efficiency of an application
- Integration of so far separate distributed applications

- Distributed access to special resources
- Improvement of fault-tolerance and availability of an application

Chapter 2

Software Architecture

Now that we have got an idea about what distributed systems are, why they exists and which goals they try to achieve, we will this subject for a while and have a look into what *software architecture* is, explain fundamental aspects of software architecture in general, emphasizing the aspects that are especially important for distributed systems.
For more information see [8].

2.1 Architecture Views

Software architecture can be viewed from two main perspectives:

External View: The view that is directly or at least indirectly visible for the user of the software

Internal View: The view that is of interest mainly for software architects and programmers.

In the following, we will focus on several points of the internal view, since the internal view exposes the architecture of the system, which is what we are interested in.
There are serveral, interlocked views on the system:

- Component View:
  Every system consists of application-centered and purely technical components.

  Application-centered components fullfill the function that the system was built for. As an example, there may be the components "customer managment", "address managment" and "date module". Some — like the "customer management" — will be visible to the user, others — e.g. the "date module" — operate invisibly in the background.

  Technical components are indepentent of the application, auch as data managment and batch control.

  This view is interesting for distributed system design, since components are the primary candidates for distribution.

- Administrative View:
  This touches questions like: How can the system be installed and de-installed? Is it possible to run several differently configured instances on one machine? How does the system react to differnt kinds of failures? ...

- "Building" View:
  Every system consists of a set of technical units that are used to create the run-time system. Those are: Source files, libraries, scripts, makefiles, executables and others.

  With a distributed system, the distribution of the software on the dedicated machines is also part of this view.

- Physical View:
  Every system uses a set of physical devices such as computers, screens and printers. In a distributed system this view is important as it concerns how the system is distributed among the computers.

- Run-Time View:
  At run-time the system exists as a set of processes (and, if applicable, threads within them) files of the machines in the system. The processes of the system can commuicate in a variety of ways (pipes, sockets, RPC (see Sect. 5.1), CORBA (see Sect.6.3.1). In every large system, the control of the processes is a central problem: What has to happen, if a certain process crashes? What processes does it affect?

This views (complemented with the external view) form the architecture of the system. Two architectures can match in severall or all points in different levels of concreteness. If two systems have the same architecture, as a rule, it is only meant that they match in a few dimensions on a low level of concreteness. As an example, the *Fat-Client-Architecture* is described by:

- Every user has her own machine (the client machine), that runs essentially the whole application (this concerns the pyhical view).

- Each of these clients communicates with a database server using some protocol (this concerns the run-time view).

2.2 What is a "good" Architecture?

We distinguish the quality of a system and the quality of its architecture. The quality of a system is determined by the quality of its architecture to a high degree, but not exclusivly. There are serveral criteria to measure the quality by, split into "hard" criteria, which are quantifiable, and "soft" ones, that are not quantifiable.
Hard quality criteria:

1. Performance:
   Crucial for the performance are the response times in the dialog with the user, they on their own, however, have no expressiveness. Performance is estimated the demanded response times and by the frequency of transactions. A transaction that is carried out only a few times a year can take hours without imposing any problems. Also important are the batch run times.

2. Security:
   Unauthirzed access can produce severy damage. A good system prevents abuse on every level: At the user interface a permission system is applied, on the machine communication level appropriate measures are taken (encryption, firewall).

3. Availability and Reliability:
   Measure of the availibility is the quotientof the times of actual and promised availibility. The mean time to failure is the measure of reliability of a system. With systems with many users one must distinguish between a failure concerning the whole system or affecting only some users or user groups.

4. Robustness:
   This is especially important for distributed systems: In the case of the failure of one machine or one component the system should be able to continue work (at least partially).

5. Range of functions

6. Usability

Each of these attributes is a non-immidiate measure of the quality of the software architecture. The architecture is good, when it is possible to precipitate every feature, like performance, security or usabililty, in every sensible degree by local patches or additions within the scope of the existing structure, at a sensible cost. The other way round, it is improbable, but not impossible that a system with a very bad architecture meets all the above points very well. However, such a system would be inflexible. *Flexibility* is an *immediate* measure of the quality of the architecture. The following ("soft") criteria affect flexibility:

1. Ability to test and integrate

2. Maintainabilty

3. Alterability

4. Portability

5. Skalability:
   This a very important criterion for distributed systems. Is it possible to get a system with 500 users to handle 5000, for example? The system should be configureable to handle more users, transactions, ... without nameable change in the application software.

6. Reusabilty

A system fullfilling all of those criteria possesses an inherently good architecture — what further demands would one have? The question remaining is what has to be done to get there.

2.2.1 Seperation of Concerns

There is no magic formula for a good architecture, but there is a obligatory rule: seperation of concerns. Software that is occupied with different things at once is bad in every aspect. The horror of every programmer is returncodes of different APIs mixed with application logic, all in a few lines of code.
This formalisation of this idea leads to a scheme of four categories for software components. Each components can be:

- independent of application and technology,
- determined by the application, independent of technology,
- independent of the application, determined by technology,
- determinded by both application and technology.

Application-determined code knows concept such as "air passenger", "airline". Technology-determined code knows at least one technical API such as ODBC or OCI.
For abbreviation we mark application-determined code as "A", technology-determined code as "T" and neutral code as "0". Thus we get the four categories "0", "A", "T" and "AT". 0-software is ideally reuseable, but makes no sense on its own. Class libraries such as the C++-STL (Standard Template Library) are examples for 0-software.
A-software can be reused whenever the present application logic is required in part or as a whole. The technical approach to reuse can be quite different, depending on the situation: A-software can be linked in directly, loaded at run-time (plugin) or accessed by middleware (COM, CORBA).
T-software can be reused when a new system uses the same technical component (JDBC, ODBC, MFC, ...).
AT-software is concerned with both application logic and technical components. It is difficult to maintain, resists changes and can hardly be reused and must therefore be avoided.
The proportion of AT-software is an important measure for the quality of the software-architecture: in the best case it is zero, in the worst case one. The total absence of AT-code is a convincing sign for the quality of the architecture.

Chapter 3

Distributed Software Architecture and Software Patterns

In this chapter, we will have a look at aspects of software architecture that apply especially to distributed applications. We will study *software patterns* that can be used to solve problems arising in distributed contexts.
The distributed application functionality is subject to change since it is often used in unforeseen contexts, e.g.:

- Accessed from different clients
- Run on different platforms
- Configured into different run-time contexts

The solution to this problem is to not structure the distributed applicatio as a monolith, but instead decompose it into classes, frameworks and components. This has been discussed in part already in Sect. 2.2.1.
A class is a unit of abstraction and implementation in an OO programming language (but can be modelled in non-OO languages with some effort).
A framework is an integrated collection of classes that collaborate to produce a reusable architecture for a family of related applications.
A component is an encapsulation unit with one or more interfaces that provide clients with access to its services.
Software patterns are a recent software engineering problem-solving discipline that emerged from the object-oriented community. Software patterns first became popular with the object-oriented Design Patterns book ([5]).
A pattern is abstraction from a concrete form which keeps recurring in specific, non-arbitrary contexts.
In the following we will discuss a few patterns relevant to distributed systems.

3.1 Proxy

This pattern is often used for implementing network communication in RPC or higher level frameworks (such as CORBA (see Section 6.3.1 and Java RMI (see Section 6.3.2).

The intent of the pattern is to provide a surrogate or place holder to control access to an object. This is done by generating a abstract base class for every object interface that should be remotely accessed. Then, this class is used as base class twice: Once for the real object (which may use delegation if the real object to be interfaced is already implemented and can not be changed). The other subclass implements a transparent proxy: It marshals the parameters of methods invoked on it and sends them along with an identification of the method over a network to a *skeleton object*, which demarshals the parameters and invokes the method in the real object. Then this is done with the result, passing it back to the invoking code. Fig. 5.1, which shows the RPC schema, describes this pretty well, too, when you substitute client stub with proxy and server stub with skeleton.

3.2 Observer

The observer pattern is used to define relationship between a group of objects such that whenever one object is updated all others are notified automatically.
See Fig. 3.1 for an UML diagram of the pattern.
This pattern is useful for monitoring a group of (distributed) objects and reacting on changes in them.

3.3 Chain of Responsibilities

With the "Chain of Responsibilities" pattern, coupling the sender of a request to its receiver is avoided by giving more than one object a chance to handle the request.
This is solved by chaining the receiver objects in a linked list and pass the request along the chain until an object handles it.
Figure 3.2 shows this pattern.

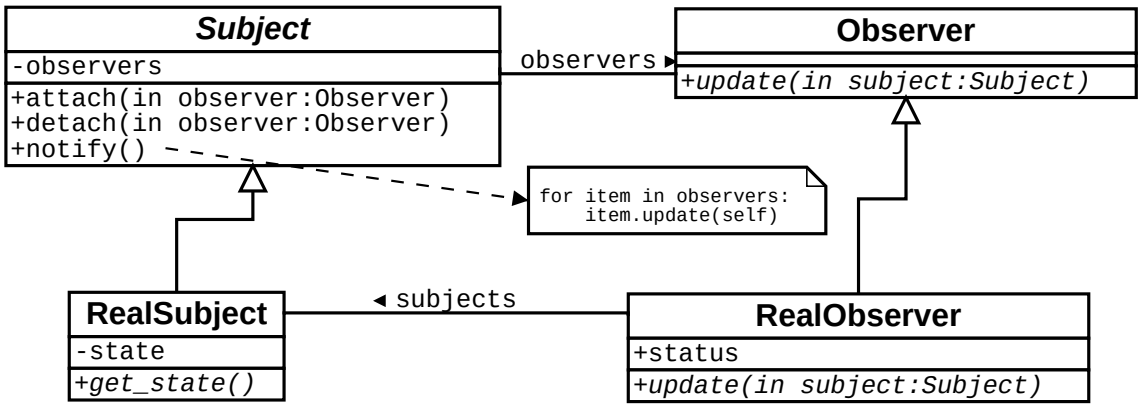

Figure 3.1: The Observer Pattern

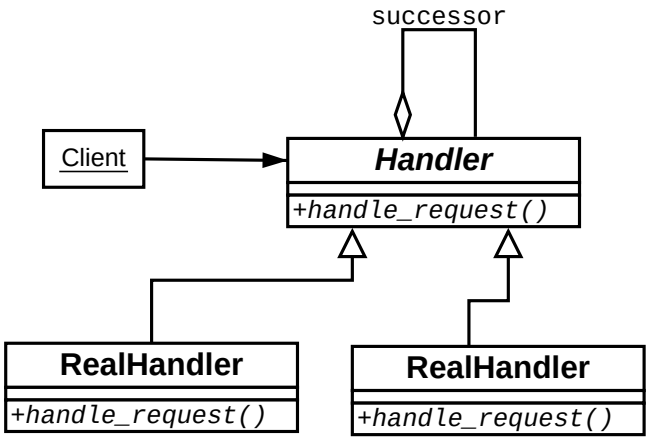

Figure 3.2: The "Chain of-Responsibilities" Pattern

Chapter 4

Distribution Models

4.1 Client-Server Model

In the Client/Server model, the software system is split between the server and the client. The client sends requests according to some protocol, asking for information or action, and the server responds.
The protocol used may range from simple message passing over TCP or UDP sockets, to RPCs (see Sect. 5.1) and higher level abstractions provided by CORBA (see Sect. 6.3.1) and Java RMI (see Sect. 6.3.2).
There may be either one centralized server or several distributed ones. This model allows clients and servers to be placed independently on nodes in a network, possibly on different hardware and operating systems appropriate to their function, e.g. fast server/cheap client. Some example configurations for a client-server architecture are listed in Table 4.1.

Name	**Client-Side Functionality**	**Server-Side Functionality**
Database Server	User Interface Application Logic	RDMBS
Application Server	User Interface	Application Logic
X Window System	Application Logic Display Access Primitives	Display Hardware Access

Table 4.1: Example Client Server Configurations

4.2 Multi-Tier Model

Multi-Tier model is an generalization of the Client/Server model, which is in fact a 2-tiered architecture. In an n-tier architecture the software system is broken down into n parts, each on a different machine.
In Fig. 4.1, the tiers composing a multi-tiered system are depicted. Each tier, except the first and the last acts both as a server, offering services to the tier above it, and as a client, using the services of the tier below it.
A typical example of an 3-tier architecture is a system that consists of a Database Server, an Application Server (accessing the Database) and a Client, presenting the user with a

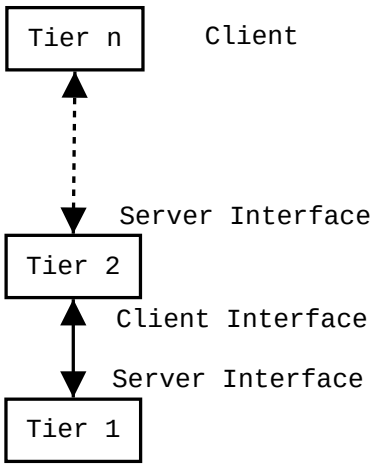

Figure 4.1: A multi-tiered system

GUI (Graphical User Interface) to the Application.
In general, the purpose of a multi-tiered architecture is to reach improvements of

Scalability: Each tier can be upgraded to a faster machine, or even be a cluster of machines independently of and without affecting the others

Modularity: Each tier of the whole system can be developed independently of the others

4.3 Peer-To-Peer Model

The model of the *peer-to-peer* architecture differs significantly from both Client/Server (see Sect. 4.1 and multi-tiered (see Sect. 4.2) architecture models.
In a peer-to-peer application, the software is distributed among machines that all have the same capabilities and responsibilities.

Chapter 5

Distributed Software Architecture Concepts

In this section, some basic concepts of distributed system architecture are explored, as a understanding of them is necessary for the later sections.

5.1 Remote Procedure Calls

Remote procedure calls (RPC) are a widely employed mechanism to support programming distributed applications.
It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

5.1.1 How an RPC works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. In this context, the caller is often called *client* and the process implementing the procedure is often called *server*.
Since the called procedure can be on another host and certainly is in another address space, there have to be means of transfer the arguments into the address space of the called process and transfer the result of the procedure back. This generally involves a transport over a network connection (which maybe via the *loopback interface*[1] if the two processes are on the same host). Thus the data passed between the processes participating in the RPC must be transformed to an "on-the-wire" format when sent and back into the host-specific format when received. This process is called *marshaling*. To make an RPC call look similar to a local procedure call, the marshaling and unmarshaling is done

[1]The interface that allows "network" connections between processes on the same host

transparently (or rather nearly transparent) by entities called *stubs* in both the client and the server process. The protocol of an RPC is illustrated in Fig. 5.1.

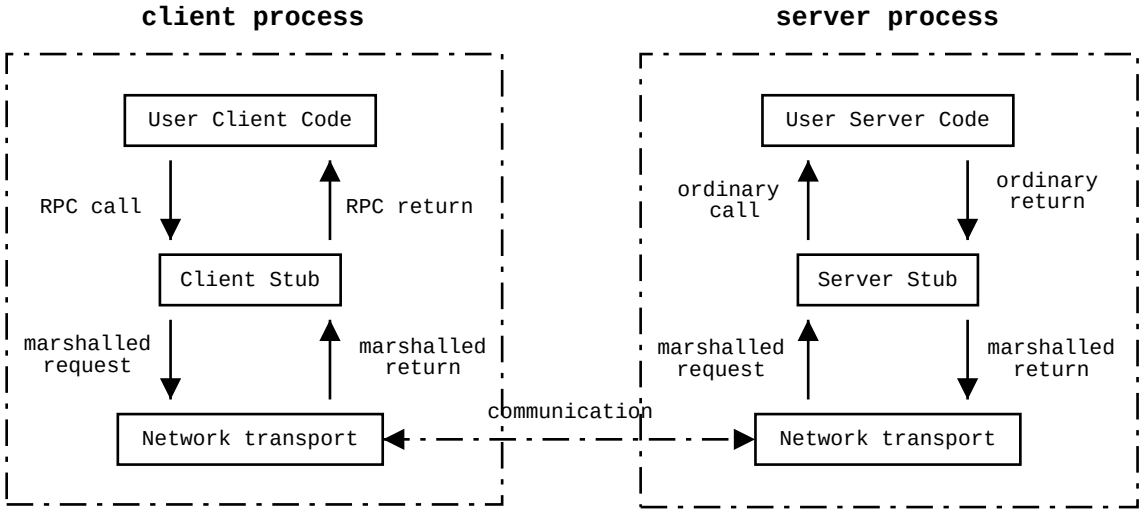

Figure 5.1: The protocol of an RPC invocation

Chapter 6

Approaches to Distribution

In this section, the fundamental approaches to the architecture of a distributed system are briefly discussed.

6.1 Distributed Operating System Approach

Quite early in their history distributed systems attracted the attention of operating-system specialists. The implementation of the operating system as a distributed program, i.e. as a *distributed operating system* is aimed at presenting the distributed system as a single virtual machine.
This approach tries to take the burden of distribiution off the application programmer, by making distribution transparent to application programs. However, since the unit of distribition in the distributed OS will be the process, applications not specifically designed to run as set of processes will not achieve any performance gain, altough the overal system performance will generally be higher due to possible load balancing.
Here we will have a brief look at an implementation of an distributed OS layer on top of the free Linux kernel, called MOSIX.

6.1.1 MOSIX

MOSIX [1] is a software that can transform a cluster of PC's (workstations and servers) to run almost like an SMP. The main advantages of an SMP are simplicity of use and near optimal resource usage, i.e., you need only to execute (one or more) processes without worrying too much where they run. MOSIX does exactly the same. You can create many processes in one node and MOSIX will distribute the processes (transparently) among the nodes, to get the best performance.
MOSIX works by migrating the user level part of processes according to a load distribution algorithm. The system level and a so called *deputity* remain on the node where the process was started. The migration is made tranparent by delegating all requests to resources on the *home node* (i.e. the original host) to the deputy.
The adaptive load distribution algorithm is descentral, allowing dynamical addition and removal of nodes to the system.
However, there are applications that cannot be migrated off the home node. This is the case if an application uses shared or locked memory, accesses the hardware directly or

uses multiple threads. This restricts the usability of MOSIX in practice quite a lot: Many appliances such as Apache or Zope and some end-user programs like Netscape or StarOffice cannot be migrated.

6.2 Database Approach

The cooperation between processes of a distributed application basicly is composed of *communication* and *synchronization*. Synchronization can be viewed as a special case of communication, where the exchanged information is quasi 1 bit (with two partners). Therefore the interaction of processes in a distributed application can be viewed as communication, that is the exchange of information and thus the ordered joint access of information by several (independent) partners.
Exactly this information access is provided by databases, in distributed environments also distributed databases. So it would be feasible to build a distributed application out of autonomous, traditional sequential processes that know nothing of each other. Cooperation is implicitly achieved via access to the shared database.

6.3 Programming Language and Middleware-Based Approach

In the following several specific implementations of this approach are discussed.

6.3.1 CORBA

CORBA (Component Object Request Broker Architecture) [2]), published by OMG (Open Management Group) [3] is an open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can inter-operate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.
CORBA applications are composed of objects. For each object type, there is an interface definition in OMG IDL (Interface Definition Language). The IDL syntax is somewhat similar to C++. The interface is the syntax part of the contract that the server object offers to the clients that invoke it. Any client that wants to invoke an operation on the object must use this IDL interface to specify the operation it wants to perform, and then basically an RPC (see Sect. 5.1) is carried out to invoke the method on the remote object. The IDL interface definition is independent of programming language, but maps to all of the popular programming languages via OMG standards: OMG has standardized mappings from IDL to C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript.
This separation of interface from implementation, enabled by OMG IDL, is the essence of CORBA - how it enables interoperability, with all of the transparencies mentioned above. The interface to each object is defined very strictly. In contrast, the implementation of an object - its running code, and its data — is hidden from the rest of the system (that is, encapsulated) behind a boundary that the client may not cross. Clients access objects

only through their advertised interface, invoking only those operations that that the object exposes through its IDL interface, with only those parameters (input and output) that are included in the invocation.
The figure detailing the RPC invocation protocol (Fig. 5.1) matches what CORBA does quite well. However, there is an additional layer below the stubs and above the network transport, called ORB (Object Request Broker), that handles object identification, request routing, object activation (it is possible to "suspend" an object and have it activated when a request comes in) and related issues. For details on the ORB see [4]. Furthermore server stubs are called skeletons in CORBA terminology.
Client Stubs and Server Skeletons are both compiled from the IDL interface definition. Because IDL defines interfaces strictly, the stub on the client side has no trouble meshing perfectly with the skeleton on the server side, even if the two are compiled into different programming languages, or even running on different CORBA implementations of different vendors.
In CORBA, every object instance has its own unique object reference, an identifying electronic token. Clients use the object references to direct their invocations, identifying to the ORB the exact instance they want to invoke.

6.3.2 Java RMI

The popular *Java* [6] programming language has its own native mechanism for distributing objects, called *RMI* (Remote Method Invocation, see [7]), although there is also support for CORBA.
RMI extends the Java object model beyond a single virtual machine address space. Object methods can be invoked between different VMs across a network, and actual objects can be passed as arguments and return values during method invocation. Java RMI uses object serialization to convert object graphs to byte-streams for transport. Any Java object type can be passed during invocation. Java RMI could be described as a natural progression of procedural RPC (see Sect. 5.1), adapted to an object-oriented paradigm.
Because Java RMI can dynamically resolve method invocations across VM boundaries, it provides a fully object-oriented (OO) distributed environment. Developers can implement classic OO design patterns for distributed programming just as they would in local programming.
This removes a great deal of complexity. Unlike language-neutral object models, RMI requires no mapping to common interface definition languages. The syntax of remote method invocations is almost exactly the same as local method invocations. RMI removes the burden of memory management from the programmer, because the underlying system provides distributed garbage collection.
RMI also exhibits some distinctive new capabilities. Executable code can be dynamically distributed on demand, including all necessary code for distributed applications (client objects, remote interfaces, and remote object stubs). This means that no code needs to be preinstalled on client machines, greatly reducing the burdens of software distribution and system maintenance. Because the common type system is the Java VM and language environment, RMI works reliably across different operating systems, wherever a Java-compatible VM is available. The RMI system also takes advantage of the secure nature of the Java environment.

Appendix A

Questions & Answers

1. [**Q**] Define the term "Distributed System".
   [**A**] A distributed system is a set of physically separate processors connected by one or more communication links.

2. [**Q**] Explain the difference between *loosely* and *tightly coupled* systems.
   [**A**] In a tightly coupled system the processors share clock and memory and run one operating system. In a loosely coupled system, they have their own clock and memory and each runs its own instance of an operating system.

3. [**Q**] Name three goals of distributing software.
   [**A**] Choose three of:

   - Decentralization of data and functions of an overall application
   - Cooperation of distributed processing units
   - Improvement of locality-properties and efficiency of an application
   - Integration of so far separate distributed applications
   - Distributed access to special resources
   - Improvement of fault-tolerance and availability of an application

4. [**Q**] Software architecture can be viewed from an external and an internal point of view. Name three "internal views" of a software architecture.
   [**A**] Choose three of:

   - Component View
   - Administrative View
   - "Building" View
   - Physical View
   - Run-Time View

5. [**Q**] There are several hard criteria to measure the quality of a software architecture by. Name three of them.
   [**A**] Choose three of:

   - Performance

- Security
- Availability and Reliability
- Robustness
- Range of Functions
- Usability

6. [**Q**] Define "framework".
   [**A**] A framework is an integrated collection of classes that collaborate to produce a reusable architecture for a family of related applications.

7. [**Q**] Define "component".
   [**A**] A component is an encapsulation unit with one or more interfaces that provide clients with access to its services.

8. [**Q**] Shortly describe the Client/Server model.
   [**A**] In the Client/Server model, the software system is split between the server and the client. The client sends requests according to some protocol, asking for information or action, and the server responds.

9. [**Q**] Formulate the basic concept of the *peer-to-peer* model.
   [**A**] In a peer-to-peer application, the software is distributed among machines that all have the same capabilities and responsibilities.

10. [**Q**] Shortly explain the term "separation of concerns" in the context of software architecture. [**A**] "Separation of conecerns" means that a [**A**] The speed of digital communication (e.g. LAN) has grown faster than the speed of the computers. Thus the cost of distributing software (causing

Bibliography

[1] MOSIX, `http://www.mosix.org`.

[2] OMG CORBA FAQ, `http://www.omg.org/gettingstarted/corbafaq.htm`.

[3] OMG Home, `http://www.omg.org`.

[4] OMG ORB Basics, `http://www.omg.org/gettingstarted/orb_basics.htm`.

[5] Gamma et al. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[6] Sun Microsystems, Inc. `http://java.sun.com` - The Source for Java™ Technology.

[7] Sun Microsystems, Inc. RMI and Java™ Distributed Computing, `http://java.sun.com/features/1997/nov/rmi.html`.

[8] Siedersleben und Meyer. Software architektur. *Univ. Sbg. CS SE II PS Skriptum*, unknown.

[9] Mühlhauser und Schill. *Software Engineering für verteilte Anwendungen.* Springer Verlag, 1992.