be available to all the subscribers that registered for the corresponding event. There are two major strategies for dispatching the event to the subscribers:

- *Push strategy*. In this case it is the responsibility of the publisher to notify all the subscribers—for example, with a method invocation.
- *Pull strategy*. In this case the publisher simply makes available the message for a specific event, and it is responsibility of the subscribers to check whether there are messages on the events that are registered.

The publish-and-subscribe model is very suitable for implementing systems based on the one-to-many communication model and simplifies the implementation of indirect communication patterns. It is, in fact, not necessary for the publisher to know the identity of the subscribers to make the communication happen.

Request-reply message model

The request-reply message model identifies all communication models in which, for each message sent by a process, there is a reply. This model is quite popular and provides a different classification that does not focus on the number of the components involved in the communication but rather on how the dynamic of the interaction evolves. Point-to-point message models are more likely to be based on a request-reply interaction, especially in the case of direct communication. Publish-and-subscribe models are less likely to be based on request-reply since they rely on notifications.

The models presented here constitute a reference for structuring the communication among components in a distributed system. It is very uncommon that one single mode satisfies all the communication needs within a system. More likely, a composition of modes or their conjunct use in order to design and implement different aspects is the common case.

2.5 Technologies for distributed computing

In this section, we introduce relevant technologies that provide concrete implementations of interaction models, which mostly rely on message-based communication. They are remote procedure call (RPC), distributed object frameworks, and service-oriented computing.

2.5.1 Remote procedure call

RPC is the fundamental abstraction enabling the execution of procedures on client's request. RPC allows extending the concept of a procedure call beyond the boundaries of a process and a single memory address space. The called procedure and calling procedure may be on the same system or they may be on different systems in a network. The concept of RPC has been discussed since 1976 and completely formalized by Nelson [111] and Birrell [112] in the early 1980s. From there on, it has not changed in its major components. Even though it is a quite old technology, RPC is still used today as a fundamental component for IPC in more complex systems.

Figure 2.14 illustrates the major components that enable an RPC system. The system is based on a client/server model. The server process maintains a registry of all the available procedures that

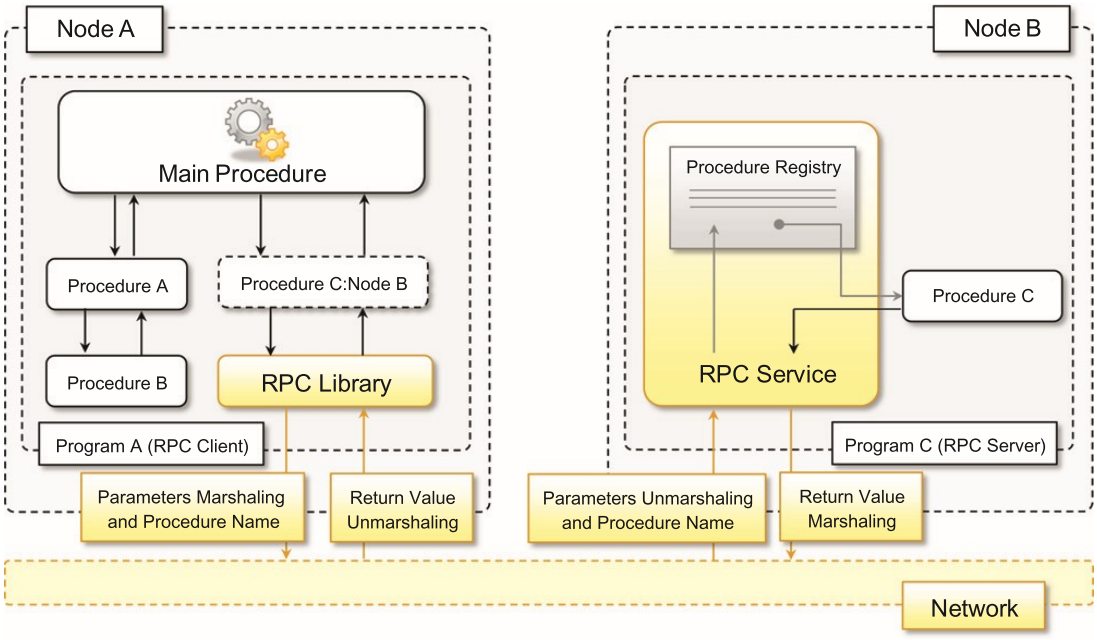

FIGURE 2.14

The RPC reference model.

can be remotely invoked and listens for requests from clients that specify which procedure to invoke, together with the values of the parameters required by the procedure. RPC maintains the synchronous pattern that is natural in IPC and function calls. Therefore, the calling process thread remains blocked until the procedure on the server process has completed its execution and the result (if any) is returned to the client.

An important aspect of RPC is *marshaling*, which identifies the process of converting parameter and return values into a form that is more suitable to be transported over a network through a sequence of bytes. The term *unmarshaling* refers to the opposite procedure. Marshaling and unmarshaling are performed by the RPC runtime infrastructure, and the client and server user code does not necessarily have to perform these tasks. The RPC runtime, on the other hand, is not only responsible for parameter packing and unpacking but also for handling the request-reply interaction that happens between the client and the server process in a completely transparent manner. Therefore, developing a system leveraging RPC for IPC consists of the following steps:

- Design and implementation of the server procedures that will be exposed for remote invocation.
- Registration of remote procedures with the RPC server on the node where they will be made available.
- Design and implementation of the client code that invokes the remote procedure(s).

Each RPC implementation generally provides client and server application programming interfaces (APIs) that facilitate the use of this simple and powerful abstraction. An important observation has to be made concerning the passing of parameters and return values. Since the server and the client processes are in two separate address spaces, the use of parameters passed by references

or pointers is not suitable in this scenario, because once unmarshaled these will refer to a memory location that is not accessible from within the server process. Second, in user-defined parameters and return value types, it is necessary to ensure that the RPC runtime is able to marshal them. This is generally possible, especially when user-defined types are composed of simple types, for which marshaling is naturally provided.

RPC has been a dominant technology for IPC for quite a long time, and several programming languages and environments support this interaction pattern in the form of libraries and additional packages. For instance, RPyC is an RPC implementation for Python. There also exist platform-independent solutions such as XML-RPC and JSON-RPC, which provide RPC facilities over XML and JSON, respectively. Thrift [113] is the framework developed at Facebook for enabling a transparent cross-language RPC model. Currently, the term RPC implementations encompass a variety of solutions including frameworks such distributed object programming (CORBA, DCOM, Java RMI, and .NET Remoting) and Web services that evolved from the original RPC concept. We discuss the peculiarity of these approaches in the following sections.

2.5.2 Distributed object frameworks

Distributed object frameworks extend object-oriented programming systems by allowing objects to be distributed across a heterogeneous network and provide facilities so that they can coherently act as though they were in the same address space. Distributed object frameworks leverage the basic mechanism introduced with RPC and extend it to enable the remote invocation of object methods and to keep track of references to objects made available through a network connection.

With respect to the RPC model, the infrastructure manages instances that are exposed through well-known interfaces instead of procedures. Therefore, the common interaction pattern is the following:

1. The server process maintains a registry of active objects that are made available to other processes. According to the specific implementation, active objects can be published using interface definitions or class definitions.
2. The client process, by using a given addressing scheme, obtains a reference to the active remote object. This reference is represented by a pointer to an instance that is of a shared type of interface and class definition.
3. The client process invokes the methods on the active object by calling them through the reference previously obtained. Parameters and return values are marshaled as happens in the case of RPC.

Distributed object frameworks give the illusion of interaction with a local instance while invoking remote methods. This is done by a mechanism called a *proxy skeleton*. Figure 2.15 gives an overview of how this infrastructure works. Proxy and skeleton always constitute a pair: the server process maintains the skeleton component, which is in charge of executing the methods that are remotely invoked, while the client maintains the proxy component, allowing its hosting environment to remotely invoke methods through the proxy interface. The transparency of remote method invocation is achieved using one of the fundamental properties of object-oriented programming: inheritance and subclassing. Both the proxy and the active remote object expose the same interface, defining the set of methods that can be remotely called. On the client side, a runtime object subclassing the type published by the server is generated. This object translates the local method invocation into an RPC call

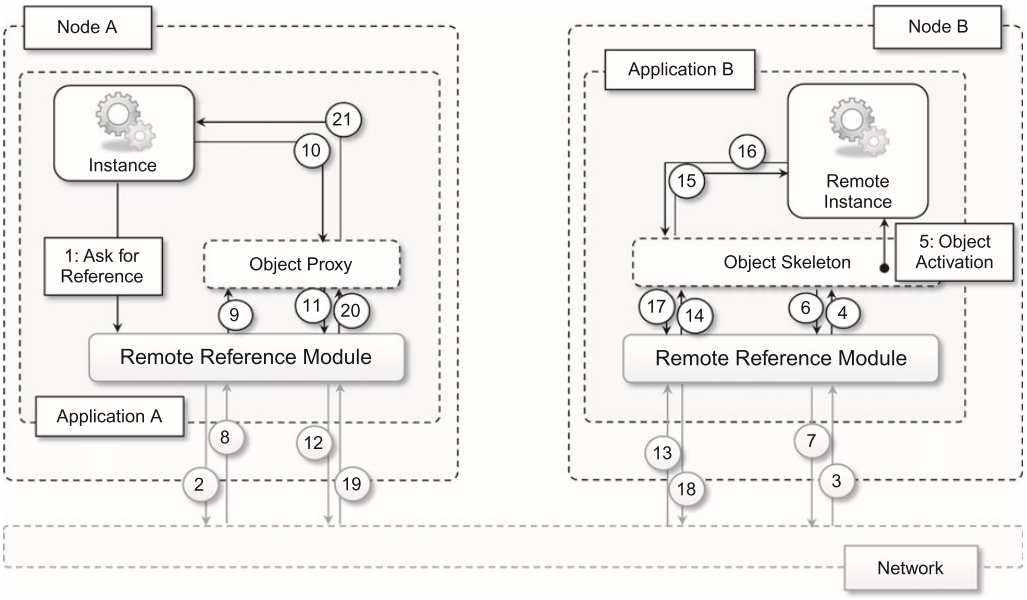

FIGURE 2.15

The distributed object programming model.

for the corresponding method on the remote active object. On the server side, whenever an RPC request is received, it is unpacked and the method call is dispatched to the skeleton that is paired with the client that issued the request. Once the method execution on the server is completed, the return values are packed and sent back to the client, and the local method call on the proxy returns.

Distributed object frameworks introduce objects as first-class entities for IPC. They are the principal gateway for invoking remote methods but can also be passed as parameters and return values. This poses an interesting problem, since object instances are complex instances that encapsulate a state and might be referenced by other components. Passing an object as a parameter or return value involves the duplication of the instance on the other execution context. This operation leads to two separate objects whose state evolves independently. The duplication becomes necessary since the instance needs to trespass the boundaries of the process. This is an important aspect to take into account in designing distributed object systems, because it might lead to inconsistencies. An alternative to this standard process, which is called *marshaling by value*, is *marshaling by reference*. In this second case the object instance is not duplicated and a proxy of it is created on the server side (for parameters) or the client side (for return values). Marshaling by reference is a more complex technique and generally puts more burden on the runtime infrastructure since remote references have to be tracked. Being more complex and resource demanding, marshaling by reference should be used only when duplication of parameters and return values lead to unexpected and inconsistent behavior of the system.

2.5.2.1 Object activation and lifetime

The management of distributed objects poses additional challenges with respect to the simple invocation of a procedure on a remote node. Methods live within the context of an object instance, and

they can alter the internal state of the object as a side effect of their execution. In particular, the lifetime of an object instance is a crucial element in distributed object-oriented systems. Within a single memory address space scenario, objects are explicitly created by the programmer, and their references are made available by passing them from one object instance to another. The memory allocated for them can be explicitly reclaimed by the programmer or automatically by the runtime system when there are no more references to that instance. A distributed scenario introduces additional issues that require a different management of the lifetime of objects exposed through remote interfaces.

The first element to be considered is the object's *activation*, which is the creation of a remote object. Various strategies can be used to manage object activation, from which we can distinguish two major classes: *server-based activation* and *client-based activation*. In server-based activation, the active object is created in the server process and registered as an instance that can be exposed beyond process boundaries. In this case, the active object has a life of its own and occasionally executes methods as a consequence of a remote method invocation. In client-based activation the active object does not originally exist on the server side; it is created when a request for method invocation comes from a client. This scenario is generally more appropriate when the active object is meant to be stateless and should exist for the sole purpose of invoking methods from remote clients. For example, if the remote object is simply a gateway to access and modify other components hosted within the server process, client-based activation is a more efficient pattern.

The second element to be considered is the lifetime of remote objects. In the case of server-based activation, the lifetime of an object is generally user-controlled, since the activation of the remote object is explicit and controlled by the user. In the case of client-based activation, the creation of the remote object is implicit, and therefore its lifetime is controlled by some policy of the runtime infrastructure. Different policies can be considered; the simplest one implies the creation of a new instance for each method invocation. This solution is quite demanding in terms of object instances and is generally integrated with some lease management strategy that allows objects to be reused for subsequent method invocations if they occur within a specified time interval (lease). Another policy might consider having only a single instance at a time, and the lifetime of the object is then controlled by the number and frequency of method calls. Different frameworks provide different levels of control of this aspect.

Object activation and lifetime management are features that are now supported to some extent in almost all the frameworks for distributed object programming, since they are essential to understanding the behavior of a distributed system. In particular, these two aspects are becoming fundamental in designing components that are accessible from other processes and that maintain states. Understanding how many objects representing the same component are created and for how long they last is essential in tracking inconsistencies due to erroneous updates to the instance internal data.

2.5.2.2 Examples of distributed object frameworks

The support for distributed object programming has evolved over time, and today it is a common feature of mainstream programming languages such as C# and Java, which provide these capabilities as part of the base class libraries. This level of integration is a sign of the maturity of this technology, which originally was designed as a separate component that could be used in several programming languages. In this section, we briefly review the most relevant approaches to and technologies for distributed object programming.

Common object request broker architecture (CORBA)

CORBA is a specification introduced by the Object Management Group (OMG) for providing cross-platform and cross-language interoperability among distributed components. The specification was originally designed to provide an interoperation standard that could be effectively used at the industrial level. The current release of the CORBA specification is version 3.0 and currently the technology is not very popular, mostly because the development phase is a considerably complex task and the interoperability among components developed in different languages has never reached the proposed level of transparency. A fundamental component in the CORBA architecture is the *Object Request Broker (ORB)*, which acts as a central object bus. A CORBA object registers with the ORB the interface it is exposing, and clients can obtain a reference to that interface and invoke methods on it. The ORB is responsible for returning the reference to the client and managing all the low-level operations required to perform the remote method invocation. To simplify cross-platform interoperability, interfaces are defined in *Interface Definition Language (IDL)*, which provides a platform-independent specification of a component. An IDL specification is then translated into a *stub-skeleton* pair by specific CORBA compilers that generate the required client (stub) and server (skeleton) components in a specific programming language. These templates are completed with an appropriate implementation in the selected programming language. This allows CORBA components to be used across different runtime environment by simply using the stub and the skeleton that match the development language used. A specification meant to be used at the industry level, CORBA provides interoperability among different implementations of its runtime. In particular, at the lowest-level ORB implementations communicate with each other using the *Internet Inter-ORB Protocol (IIOP)*, which standardizes the interactions of different ORB implementations. Moreover, CORBA provides an additional level of abstraction and separates the ORB, which mostly deals with the networking among nodes, from the *Portable Object Adapter (POA)*, which is the runtime environment in which the skeletons are hosted and managed. Again, the interface of these two layers is clearly defined, thus giving more freedom and allowing different implementations to work together seamlessly.

Distributed component object model (DCOM/COM+)

DCOM, later integrated and evolved into COM+, is the solution provided by Microsoft for distributed object programming before the introduction of .NET technology. DCOM introduces a set of features allowing the use of COM components beyond the process boundaries. A COM object identifies a component that encapsulates a set of coherent and related operations; it was designed to be easily plugged into another application to leverage the features exposed through its interface. To support interoperability, COM standardizes a binary format, thus allowing the use of COM objects across different programming languages. DCOM enables such capabilities in a distributed environment by adding the required IPC support. The architecture of DCOM is quite similar to CORBA but simpler, since it does not aim to foster the same level of interoperability; its implementation is monopolized by Microsoft, which provides a single runtime environment. A DCOM server object can expose several interfaces, each representing a different behavior of the object. To invoke the methods exposed by the interface, clients obtain a pointer to that interface and use it as though it were a pointer to an object in the client's address space. The DCOM runtime is responsible for performing all the operations required to create this illusion. This technology provides a reasonable level of interoperability among Microsoft-based environments, and there are third-party implementations that allow the use of DCOM, even in Unix-based environments. Currently, even if still used

in industry, this technology is no longer popular and has been replaced by other approaches, such as .NET Remoting and Web Services.

Java remote method invocation (RMI)

Java RMI is a standard technology provided by Java for enabling RPC among distributed Java objects. RMI defines an infrastructure allowing the invocation of methods on objects that are located on different Java Virtual Machines (JVMs) residing either on the local node or on a remote one. As with CORBA, RMI is based on the *stub-skeleton* concept. Developers define an interface extending *java.rmi.Remote* that defines the contract for IPC. Java allows only publishing interfaces while it relies on actual types for the server and client part implementation. A class implementing the previous interface represents the *skeleton* component that will be made accessible beyond the JVM boundaries. The *stub* is generated from the skeleton class definition using the *rmic* command-line tool. Once the *stub-skeleton* pair is prepared, an instance of the skeleton is registered with the RMI registry that maps URIs, through which instances can be reached, to the corresponding objects. The RMI registry is a separate component that keeps track of all the instances that can be reached on a node. Clients contact the RMI registry and specify a URI, in the form *rmi://host:port/serviceName*, to obtain a reference to the corresponding object. The RMI runtime will automatically retrieve the class information for the stub component paired with the skeleton mapped with the given URI and return an instance of it properly configured to interact with the remote object. In the client code, all the services provided by the skeleton are accessed by invoking the methods defined in the remote interface. RMI provides a quite transparent interaction pattern. Once the development and deployment phases are completed and a reference to a remote object is obtained, the client code interacts with it as though it were a local instance, and RMI performs all the required operations to enable the IPC. Moreover, RMI also allows customizing the security that has to be applied for remote objects. This is done by leveraging the standard Java security infrastructure, which allows specifying policies defining the permissions attributed to the JVM hosting the remote object.

.NET remoting

Remoting is the technology allowing for IPC among .NET applications. It provides developers with a uniform platform for accessing remote objects from within any application developed in any of the languages supported by .NET. With respect to other distributed object technologies, Remoting is a fully customizable architecture that allows developers to control the transport protocols used to exchange information between the proxy and the remote object, the serialization format used to encode data, the lifetime of remote objects, and the server management of remote objects. Despite its modular and fully customizable architecture, Remoting allows a transparent interaction pattern with objects residing on different application domains. An application domain represents an isolated execution environment that can be accessible only through Remoting channels. A single process can host multiple application domains and must have at least one.

Remoting allows objects located in different application domains to interact in a completely transparent manner, whether the two domains are in the same process, in the same machine, or on different nodes. The reference architecture is based on the classic client/server model whereby the application domain hosting the remote object is the server and the application domain accessing it

is the client. Developers define a class that inherits by *MarshalByRefObject*, the base class that provides the built-in facilities to obtain a reference of an instance from another application domain. Instances of types that do not inherit from *MarshalByRefObject* are copied across application domain boundaries. There is no need to manually generate a stub for a type that needs to be exposed remotely. The Remoting infrastructure will automatically provide all the required information to generate a proxy on a client application domain. To make a component accessible through Remoting requires the component to be registered with the Remoting runtime and mapping it to a specific URI in the form *scheme://host:port/ServiceName,* where *scheme* is generally TCP or HTTP. It is possible to use different strategies to publish the remote component: Developers can provide an instance of the type developed or simply the type information. When only the type information is provided, the activation of the object is automatic and client-based, and developers can control the lifetime of the objects by overriding the default behavior of *MarshalByRefObject*. To interact with a remote object, client application domains have to query the remote infrastructure by providing a URI identifying the remote object and they will obtain a proxy to the remote object. From there on, the interaction with the remote object is completely transparent. As happens for Java RMI, Remoting allows customizing the security measures applied for the execution of code triggered by Remoting calls.

These are the most popular technologies for enabling distributed object programming. CORBA is an industrial-standard technology for developing distributed systems spanning different platforms and vendors. The technology has been designed to be interoperable among a variety of implementations and languages. Java RMI and .NET Remoting are built-in infrastructures for IPC, serving the purpose of creating distributed applications based on a single technology: Java and .NET, respectively. With respect to CORBA, they are less complex to use and deploy but are not natively interoperable. By relying on a unified platform, both Java and .NET Remoting are very straightforward and intuitive and provide a transparent interaction pattern that naturally fits in the structure of the supported languages. Although the two architectures are similar, they have some minor differences: Java relies on an external component called *RMI registry* to locate remote objects and allows only the publication of interfaces, whereas .NET Remoting does not use a registry and allows developers to expose class types as well. Both technologies have been extensively used to develop distributed applications.

2.5.3 Service-oriented computing

Service-oriented computing organizes distributed systems in terms of *services*, which represent the major abstraction for building systems. Service orientation expresses applications and software systems as aggregations of services that are coordinated within a *service-oriented architecture (SOA)*. Even though there is no designed technology for the development of service-oriented software systems, Web services are the *de facto* approach for developing SOA. Web services, the fundamental component enabling cloud computing systems, leverage the Internet as the main interaction channel between users and the system.

2.5.3.1 What is a service?

A *service* encapsulates a software component that provides a set of coherent and related functionalities that can be reused and integrated into bigger and more complex applications. The term