Comparative Analysis between RPC and CORBA Architecture Styles
Sameer S. Yeolekar

**Purpose – General Description**

It is a tool for developing distributed, client/server-based applications that increases interoperability, portability, and flexibility of an application by allowing it to be distributed over multiple heterogeneous platforms.

It extends the notion of conventional or local procedure calling, so that the called procedure need not exist on the same machine as the calling procedure. It reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the underlying details of the various operating system and network interfaces.

**Technical Details - What does the technology do?**

To access the remote server special functional calls, called RPCs, are embedded in the client part of the client/server application program. When the client program is compiled, local stubs for client and server are created which are invoked when the application requires a remote function. It typically supports synchronous calls (also called request-reply or call/wait) between clients and servers which imply blocking of the client until the server fulfills its request.

RPC increases the flexibility of architecture because it allows the remote component to be accessed without knowledge of the network address or any other lower-level information. Most RPCs also use an asynchronous (also called call/no wait) implementations.

**Usage Considerations - Use of technology**

RPC is ideally appropriate for client/server applications in which the client issues a request and waits for the server's response before continuing its own processing (synchronous calls).

Synchronous and asynchronous call mechanisms both have individual strengths and weaknesses that should be considered when designing any specific application. The synchronous request-reply mechanism requires that client and server are always available and functioning i.e. they are not blocked. Now, to recover from a blocked condition, an RPC implementation is required to provide mechanisms such as error messages, request timers, retransmissions, or redirection to an alternate server. The complexity of the application using a RPC is dependent on the sophistication of the specific RPC implementation (i.e., the more sophisticated the recovery mechanisms supported by RPC, the less complex the application utilizing the RPC is required to be). RPCs that implement asynchronous mechanisms are very few and are difficult (complex) to implement.

When using RPC over a distributed network the performance (or load) of the network is crucial. In RPC, unlike the asynchronous mechanism, the synchronous blocking mechanism guards against network overloading. If recovery mechanisms, such as retransmissions, are employed by an RPC application, the resulting load on a network may increase, making the application inappropriate for a congested network. RPCs use static routing tables established at compile-time, the ability to perform load balancing across a network is difficult and should be considered when designing an RPC-based application.

**Maturity - How well-developed technology is?**

RPC applications development tools are available over large variety of platforms. Some of them include Windows (3.1, NT, 95), OS/2 etc. The RPC infrastructures are
implemented within the Distributed Computing Environment (DCE) and within Open Network Computing (ONC), developed by Sunsoft, Inc. The DCE infrastructure supports the construction and integration of client/server applications while attempting to hide the inherent complexity of the distributed processing from the user. These two RPC implementations dominate the current Middleware market. 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.

Costs and Limitations - Constraints of technology

Usage of a single RPC implementation in a system results in dependence on the RPC vendor for maintenance support and future enhancements. This affects the system's flexibility, maintainability, portability, and interoperability.

Since, there is no one single standard for implementing an RPC, different varied features may be offered by individual RPC implementations. These features which might affect the design and cost of a RPC-based application are as follows:

- support of synchronous and/or asynchronous processing
- support of different networking protocols
- support for different file systems
- whether the RPC mechanism can be obtained individually, or only bundled with a server operating system

Common Object Request Broker Architecture (CORBA)

Purpose – General Description

The Common Object Request Broker Architecture (CORBA) is a specification of a standard architecture for object request brokers (ORBs). ORB is a middleware technology that manages communication and data exchange between objects. It promotes the interoperability of distributed object systems by enabling users to build systems by piecing together objects from different vendors that communicate with each other via the ORB. The developers are only concerned with the object interface details rather than the implementation details. This information hiding enhances system maintainability since the object communication details are hidden from the developers and isolated in the ORB. A standard architecture such as CORBA allows vendors to develop ORB products that support application portability and interoperability across different programming languages, hardware platforms, operating systems, and ORB implementations.

Using CORBA, a client can transparently invoke a method on a server object, which can be on the same machine or on a remote host. The ORB is responsible for finding an object that can implement the request, passing it the parameters, invoking its method, and returning the results of the invocation. The client does not have to be aware of where the object is located, its programming language, its operating system or any other aspects that are not part of an object's interface.

In CORBA distributed systems are conceived and implemented as distributed objects. The interfaces to these objects are described in a high-level, architecture-neutral specification language that also supports object-oriented design abstraction.

The CORBA specification was developed by the Object Management Group (OMG), an industry group with over six hundred member companies representing computer manufacturers, independent software vendors, and a variety of government and academic organizations.

Technical Details - What does the technology do?

CORBA can be thought of as a generalization of remote procedure call
(RPC) that includes a number of refinements of RPC, including:

- a more abstract and powerful interface definition language
- direct support for a variety of object-oriented concepts
- a variety of other improvements and generalizations of the more primitive RPC

To understand CORBA we should understand the role it plays in the Object Management Architecture (OMA). The OMA is itself a specification that defines a broad range of services for building distributed applications. OMA services are partitioned into three categories:

- CORBA Service
- CORBA Facilities
- Application Objects

The ORB is used for communication between applications and these services. CORBA Services, CORBA Facilities, and Application Objects define different categories of objects in the OMA which are used for the development of distributed software systems.

- CORBA Services are considered fundamental to building non-trivial distributed applications. Services Example: Naming Service, Concurrency Service, Relationship Service
- CORBA Facilities may be useful for distributed applications in some settings, but are not considered as universally applicable as CORBA Services. Facilities Example: User Interface, Task Management.
- Application Objects provide services that are particular to an application or class of applications.

In CORBA all objects are defined IDL (Interface Definition Language). It is object-oriented with some similarities to C++ and can only define interfaces. It is not possible to specify behavior in IDL. The IDL interface defines the interface between clients and objects. All other details about objects such as their implementation language and their location are made transparent.

Usage Considerations - Use of technology

Compliance: CORBA is a specification and not an implementation so the question of compliance arises. How does an user know that a particular application is CORBA-compliant? The compliance is defined by OMG.

The CORBA Core is defined for compliance as including the following:

- the interfaces to all of the elements
- interfaces to the interface repository
- a definition of IDL syntax and semantics
- the definition of the object model that underlies CORBA (e.g., what is an object, how is it defined, where do they come from)

Till date there are no defined test suites for assessing CORBA compliance. The users should assess the likelihood of vendor compliance based upon the role played by the vendor in the OMG, vendor market share, press releases and testimonials. Also, hands-on evaluation of ORB products is an absolute necessity.

Complexity: A number of factors are responsible for the complexity of the CORBA specification:

- The ORBs are vendor dependent with different features and capabilities. This requires the users to learn the vendor implemented specification.
- The issue of building robust distributed systems is also a complexity factor.
- Deep expertise in related technologies is required.

Stability: There is great instability in the distributed object technology market. Recent advancement in distributed computing creates a need to alter the CORBA specification.
Maturity—How well-developed technology is?

A large and growing number of implementations of CORBA are available. This also includes implementations from most major computer manufacturers and independent software vendors. All CORBA ORBs are not mature. The CORBA and OMA products are still maturing and been used in complex and demanding situations but, the specification and product implementations are not entirely stable.

Costs and Limitations—Constraints of technology

- **Real time**: The CORBA version 2.0 does not address real-time issues.
- **Programming language support**: IDL is a "least-common denominator" language. It does not fully exploit the capabilities of programming languages to which it is mapped, especially where the definition of abstract types is concerned.
- **Pricing and licensing**: The price of ORBs varies greatly, from a few hundred to several thousand dollars. The Licensing schemes also vary.
- **Training**: Training for the already experienced programmer: five days of hands-on training for CORBA programming fundamentals is essential.
- **Security**: CORBA specifies only a minimal range of security mechanisms.

Dependencies—Influence of other technologies

- CORBA depends on TCP/IP to support the CORBA-defined inter-ORB interoperability protocol (IIOP).
- Most commercial CORBA ORBs rely on C++ as the principal client and server programming environment. Java-specific ORBs are emerging.

Similarities between RPC and CORBA:

<table>
<thead>
<tr>
<th>Attributes</th>
<th>RPC</th>
<th>CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules</td>
<td>Client and server stubs.</td>
<td>Client Stubs and Server Skeletons.</td>
</tr>
<tr>
<td></td>
<td>Performs marshalling/unmarshalling of data.</td>
<td>Performs marshalling/unmarshalling of data.</td>
</tr>
<tr>
<td>Definition of Types</td>
<td>Remote Procedure Call Language (RPCL)</td>
<td>Interface Definition Language (IDL)</td>
</tr>
<tr>
<td>Mechanism for mapping client’s request to server</td>
<td>Portmapper</td>
<td>Object Request Broker (ORB)</td>
</tr>
<tr>
<td>Synchronous Communication</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Asynchronous Communication</td>
<td>Capable</td>
<td>Capable</td>
</tr>
</tbody>
</table>
### Differences between RPC and CORBA:

<table>
<thead>
<tr>
<th>Attributes</th>
<th>RPC</th>
<th>CORBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>Sun Microsystems</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>Architecture Style</td>
<td>Infrastructure</td>
<td>Specification</td>
</tr>
<tr>
<td>Works on</td>
<td>Procedures</td>
<td>Objects</td>
</tr>
<tr>
<td>Target Language</td>
<td>RPCgen outputs C code</td>
<td>IDL maps to any language defined for C, C++, ADA, Smalltalk, COBOL</td>
</tr>
<tr>
<td>Size</td>
<td>Lightweight. RPC library that implements the RPC interface only consists of small number of operations and the stub generated are fairly small.</td>
<td>CORBA application tend to be much bigger.</td>
</tr>
<tr>
<td>Portmapper Vs. ORB</td>
<td>Portmapper runs on a specific host</td>
<td>ORB can hide the server’s identity from the client</td>
</tr>
<tr>
<td>Interface Repository</td>
<td>NO. Hence, the clients cannot perform a type check to ensure that a target object can support the request made by the client.</td>
<td>YES</td>
</tr>
<tr>
<td>Definition of Remote Components</td>
<td>RPCL is not very expressive as IDL since it lacks concepts of inheritance, attributes and application specific exceptions.</td>
<td>Easier using IDL as IDL is more expressive.</td>
</tr>
<tr>
<td>Support services</td>
<td>None. Its just remote procedures</td>
<td>Horizontal service: naming, events, etc. Vertical service: financial, health care, etc.</td>
</tr>
<tr>
<td>Failure Handling</td>
<td>Null Pointer returned from a remote procedure call indicate that the procedure has not been executed properly.</td>
<td>An exception is thrown which gives a precise amount of information as why an operation execution request was not handled properly.</td>
</tr>
<tr>
<td>Costs</td>
<td>Cheaper (atleast on UNIX work stations). If you have installed an UNIX operating system it will have RPCs on it. They will be just there.</td>
<td>Price of ORBs varies greatly. You have to buy CORBA products in order to use it.</td>
</tr>
</tbody>
</table>

### References