Distributed-Object Computing Tools

INTRODUCTION

Distributed computing systems are omnipresent in today’s world. The rapid progress in the semiconductor and networking infrastructures have blurred the differentiation between parallel and distributed computing systems and made distributed computing a workable alternative to high-performance parallel architectures. However attractive distributed computing may be, developing software for such systems is hardly a trivial task. Many different models and technologies have been proposed by academia and industry for developing robust distrib- uted software systems. Despite a large number of such systems, one fact is clear that the software for distributed computing can be written efficiently using the principles of distributed-object computing. The concept of objects residing in different address spaces and communicating via messages over a network meshes well with the principles of distributed computation. Out of the exist- ing alternatives, Java-RMI (Remote Method Invocation) from Sun Microsys- tems, CORBA (Common Object Request Broker Architecture) from Object Management Group, and DCOM (Distributed Component Object Model) from Microsoft are the most popular distributed-object models among researchers and practitioners.

RMI

Java Remote Method Invocation allows the implementation of distributed objects in Java. It allows a client running on any Java virtual machine to access a Java object hosted on a remote virtual machine. RMI follows the language- centric model .This fact is two-faceted. First, the clients and servers need to be implemented in Java. This gives them an inherent object-oriented appearance and allows objects to interact with all the features of Java, such as JNI and JDBC. It also means that a server object can be run and accessed from any virtual machine, thus achieving platform independency. Second, the model is tied to Java, and hence objects cannot be implemented using any other language, thereby prohibiting interactions between heterogeneous (i.e., implemented in different languages) objects.

Basic Model: The basic model of RMI consists of a client program, which intends to access a remote object, and a server program, which hosts the remote object. For a client to connect to a remote object requires a refer- ence to the object hosted by the server program. A client can locate the remote server object in two ways. These two ways differ in the manner in which the client obtains the remote reference. These are described below.

1. Explicitly obtaining a remote reference. RMI provides a nonpersistent registry called RMIREGISTRY, which should be deployed on the server machine. The server object when instantiated should register itself to the local RMIREGISTRY. This action is achieved by calling Naming.bind(). When trying to connect to a remote object, the client looks up a named instance registered at the RMIREGISTRY.

Figure 1 explains the archi- tecture and the sequence of events in a typical RMI application.

2. Implicitly obtaining a remote reference. A reference to a remote object can also be obtained as a parameter or return value in a method call. This, too, can serve as a means of accessing a remote object from a client. It is assumed that a RMI client knows which server machine the remote object is currently hosted on for it to connect so that a lookup for that object can be performed on that server machine for obtaining a reference.

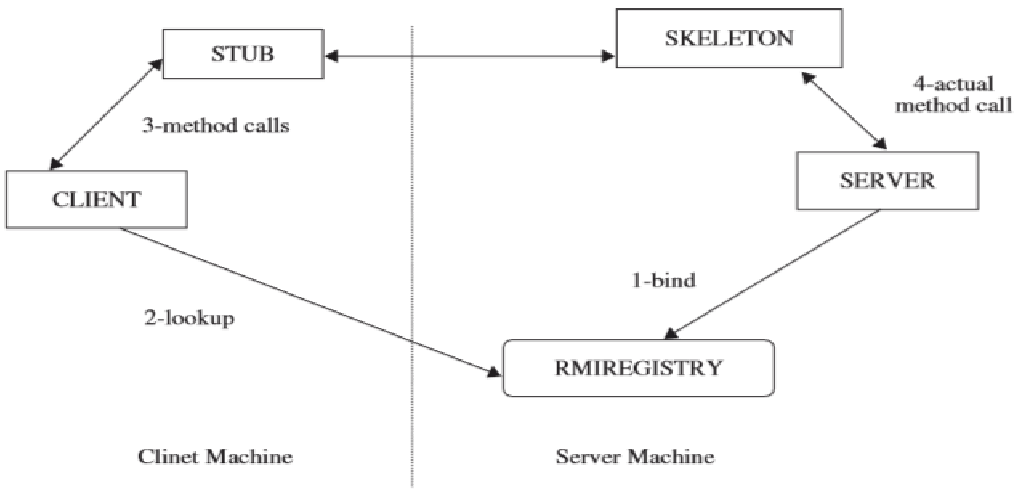


Fig.1 RMI Architecture and Sequence of Events

Irrespective of the approach used, once a remote reference is available to the client it achieves the remote method invocation using stubs and skeletons. The client-side stub acts as the proxy for the server. The server-side skeleton handles the method invocations done by all the remote stubs to the local server object. When a client invokes a method on the object reference, the call is first received by the stub. Marshaling is done by the stub and the data are sent to the server. The server-side skeleton unmarshals the method call and routes it to the actual server object. Upon completion of method execution the skeleton receives the return parameters, marshals the contents, and sends it back to the client stub, which would then unmarshal it. This entire process is transparent to the client and the method invocation looks like a local method call. The serialization ability of Java is used for the marshaling/unmar-shaling of arguments and return values.

CORBA

CORBA (Common Object Request Broker Architecture) is a distributed object architecture that provides a communication infrastructure for a heterogeneous and distributed collection of collaborating objects. These remote or local collaborating objects can interoperate across networks regardless of the language in which they are written or the platform on which they are deployed. CORBA is a suite of specifications issued by the Object Management Group (OMG) . The OMG is a nonprofit consortium comprising about 1000 computer companies. It was formed in 1989 with the purpose of pro- moting the theory and practice of object technology in distributed computing systems. OMG realizes its goals through creating standards which allow inter- operability and portability of distributed object-oriented applications. The CORBA standards for interoperability in heterogeneous computing environ-ments, promoted by the OMG, lay down guidelines for developing distributed applications whose components collaborate transparently, scalably, reliably, and efficiently .

Basic Model: CORBA Architecture

The Object Management Architecture (OMA), defined by the OMG, is a high-level design of a complete distributed environment. OMA provides the conceptual infrastructure and forms the basis for all OMG specifications, including the object model and the reference model. The object model underlies the CORBA architecture and defines common object semantics for specifying an object and its visible characteris- tics in a heterogeneous environment, and the reference model defines object interactions. The OMA comprises of four main components: object request brokers and object services (system-oriented components) and application objects and common facilities (application-oriented components) . The object request broker (ORB) is the middleware that handles the communica- tion details between the objects.

Figure 2 shows the main components of the CORBA architecture and their interconnections.

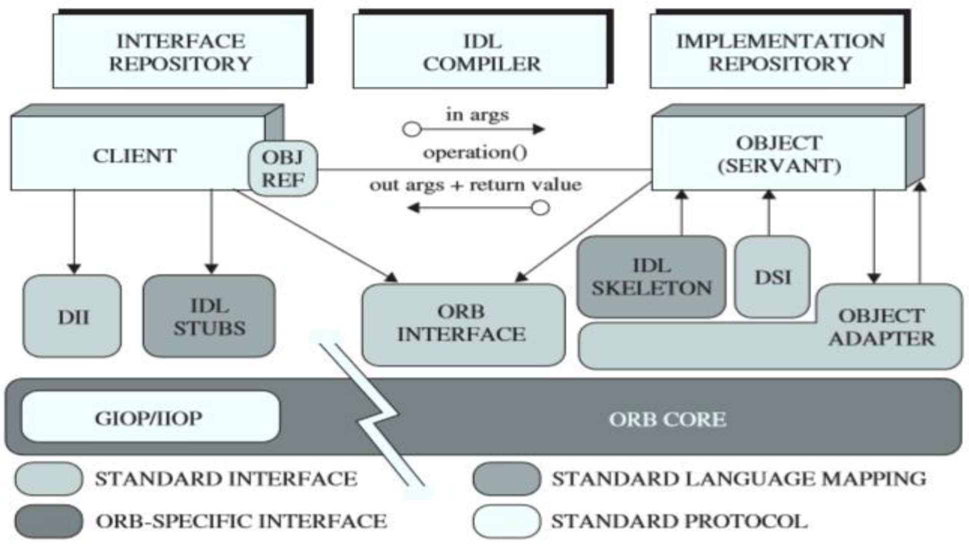


Fig. 2. CORBA Architecture

A CORBA object is a virtual entity that consists of an identity (the object reference that helps to identify, locate, and address a CORBA object), an inter-face, and an implementation. The object implementation is known as a servant. Every CORBA object maps to a servant. A servant is a programming language entity that implements the operations that are defined in the CORBA object's Interface Definition Language (IDL) interface. Servants can be implemented in various languages, such as C, C++, Java, Smalltalk, and Ada. The servants exist in the context of a server and are located and invoked by the ORB and the object adapter (OA) with the help of the object references. A client application is a program entity that invokes an operation on a remote object by maintaining a reference to the object. A client can obtain a reference to a remote server object by binding to the CORBA naming or trader service. The object request broker (ORB) is the middleware that provides a commu- nication channel for routing client requests to target object implementations transparently. A client can access the services of an object implementation only through the ORB. The ORB is responsible for finding the object implementa- tion, activating it if necessary, delivering the request to the object, and returning any response to the calling object. CORBA 2.0 specifies that each ORB must support a standard adapter called the basic object adapter (BOA). Servers may support more than one object adapter. CORBA 3.0 introduces a portable version of BOA called the portable object adapter (POA).The BOA and POA are responsible for the transparent activation of objects. POA provides a more flexible architecture that allows the ORB implementations to be designed such that the CORBA servers can fit a variety of application needs. The POA also introduces some additional features, such as providing the option of using a servant manager for each implementation of an object reference. These servant managers, also called callback objects or instance managers, assist the POA in the management of server-side objects. The ORB interface is an abstract interface defined in the CORBA specifi- cation, containing various helper functions for stringification (converting object references to strings), destringification (reverse of stringification), and creating argument lists for requests made through the dynamic invocation interface (DII). This logical ORB entity can be implemented in various ways by different vendors. A CORBA IDL stub serves as a connection between the client applications and the ORB. The IDL compiler generates the client-side stubs and the server- side skeletons. A skeleton is a programming language entity that serves as a connection between the OA and the servant and allows the OA to dispatch requests to the servant. The dynamic invocation interface (DII) allows a client application to invoke a request on a server object directly by using the underlying request mecha- nisms provided by an ORB without having to go through the IDL interface- specific stubs. The DII allows clients to make both nonblocking deferred synchronous and one-way calls.

The dynamic skeleton interface (DSI) is the server-side counterpart of the DII. The DSI allows an ORB to deliver requests to an object implementation that does not have IDL-based compiled skeletons or stubs. The object adapter acts as an intermediary between the ORB and the object implementation. The object adapter is responsible for associating object imple- mentations with the ORB, activating/deactivating the objects and delivering requests to the objects. Object implementations can support multiple object adapters. For various objects of different ORBs to be able to communicate seam- lessly with one another, the CORBA 2.0

specification provides a methodology known as the ORB Interoperability Architecture or the General Inter-ORB Protocol (GIOP). The GIOP is a collection of message requests that ORBs can make over a network. GIOP maps ORB requests to different network transports. The Internet Inter-ORB Protocol (IIOP) ,which maps GIOP messages to TCP/IP, is a GIOP implementation but a standardized version that all ORBs must be able to use. An Environment Specific Inter- ORB Protocol (ESIOP) is the complement of GIOP. ESIOPs let GIOP type messages be mapped to a proprietary network protocol such as the Open Soft-ware Foundation’s (OSF’s) Distributed Computing Environment (DCE). Any 2.0 ORB based on an ESIOP must include a half bridge to IIOP so that IIOP requests can also be made on it .

IDL (Interface Definition Language) The Interface Definition Language (IDL), defined by the OMG, is used to describe object interfaces in a standard manner. IDL is a declarative language and its grammar is an extension of a subset of the ANSI C++standard. The IDL interfaces are similar to the inter- faces in Java and abstract classes in C++. OMG provides mappings from IDL to different implementation languages such as Java and C++ . The IDL compiler generates the client-side stubs and server-side skeletons.

CORBA Object Services CORBA object services are a set of interfaces and objects used for handling the invoking and implementation of objects. Currently, CORBA provides about 15 object services. The prominent ones are:

• Naming service: helps clients find object implementations based on their name.

• Trading service: allows objects to advertise their services and bid for contracts.

• Event service: provides for an event-channel interface that distributes events among components.

Other services are persistent object service, life-cycle service, time service, transaction service, relationship service, externalization service, concurrency control service, query service, licensing service, property service, security service, and collection service.

Distributed-Object Computing Tools

Hadoop

Hadoop is an open-source framework that allows to store and process big data in a distributed environment across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

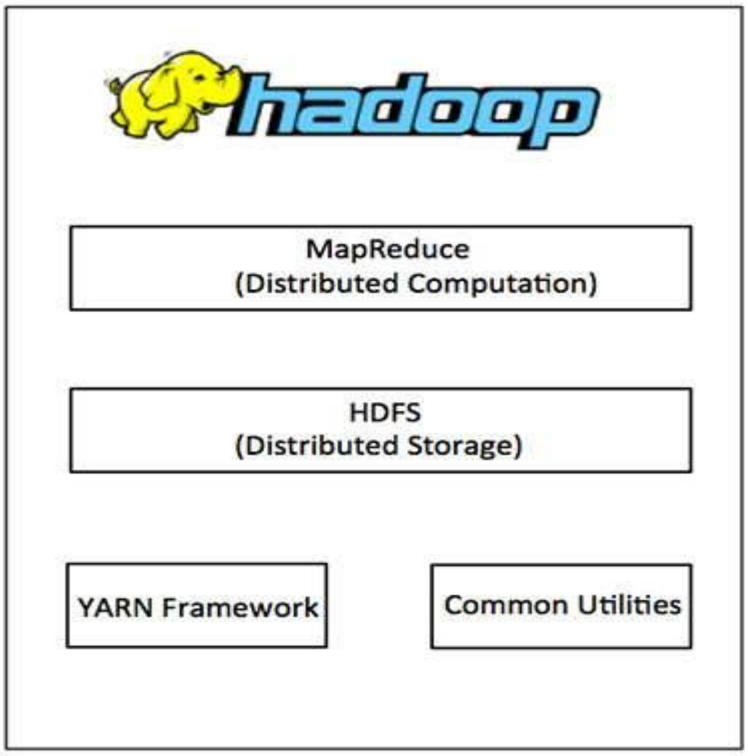
Hadoop is a distributed file system, which lets you store and handle massive amount of data on a cloud of machines, handling data redundancy. The primary benefit is that since data is stored in several nodes, it is better to process it in distributed manner. Each node can process the data stored on it instead of spending time in moving it over the network. On the contrary, in Relational database computing system, you can query data in real-time, but it is not efficient to store data in tables, records and columns when the data is huge.

Hadoop is an Apache open source framework written in java that allows distributed processing of large datasets across clusters of computers using simple programming models. The Hadoop framework application works in an environment that provides distributed *storage* and *computation* across clusters of computers. Hadoop is designed to scale up from single server to thousands of machines, each offering local computation and storage.

Hadoop Architecture

At its core, Hadoop has two major layers namely −

- Processing/Computation layer (MapReduce), and
- Storage layer (Hadoop Distributed File System).



MapReduce: MapReduce is a parallel programming model for writing distributed applications devised at Google for efficient processing of large amounts of data (multi-terabyte data-sets), on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. The MapReduce program runs on Hadoop which is an Apache open-source framework.

Hadoop Distributed File System: The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and provides a distributed file system that is designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. It is highly fault-tolerant and is designed to be deployed on low-cost hardware. It provides high throughput access to application data and is suitable for applications having large datasets.

Apart from the above-mentioned two core components, Hadoop framework also includes the following two modules −

- **Hadoop Common** − These are Java libraries and utilities required by other Hadoop modules.
- **Hadoop YARN** − This is a framework for job scheduling and cluster resource management.

Working of Hadoop

It is quite expensive to build bigger servers with heavy configurations that handle large scale processing, but as an alternative, you can tie together many commodity computers with single-CPU, as a single functional distributed system and practically, the clustered machines can read the dataset in parallel and provide a much higher throughput. Moreover, it is cheaper than one high-end server. So this is the first motivational factor behind using Hadoop that it runs across clustered and low-cost machines.

Hadoop runs code across a cluster of computers. This process includes the following core tasks that Hadoop performs −

- Data is initially divided into directories and files. Files are divided into uniform sized blocks of 128M and 64M (preferably 128M).
- These files are then distributed across various cluster nodes for further processing.
- HDFS, being on top of the local file system, supervises the processing.
- Blocks are replicated for handling hardware failure.
- Checking that the code was executed successfully.
- Performing the sort that takes place between the map and reduce stages.
- Sending the sorted data to a certain computer.
- Writing the debugging logs for each job.

Advantages of Hadoop

- Hadoop framework allows the user to quickly write and test distributed systems. It is efficient, and it automatic distributes the data and work across the machines and in turn, utilizes the underlying parallelism of the CPU cores.
- Hadoop does not rely on hardware to provide fault-tolerance and high availability (FTHA), rather Hadoop library itself has been designed to detect and handle failures at the application layer.

- Servers can be added or removed from the cluster dynamically and Hadoop continues to operate without interruption.
- Another big advantage of Hadoop is that apart from being open source, it is compatible on all the platforms since it is Java based.

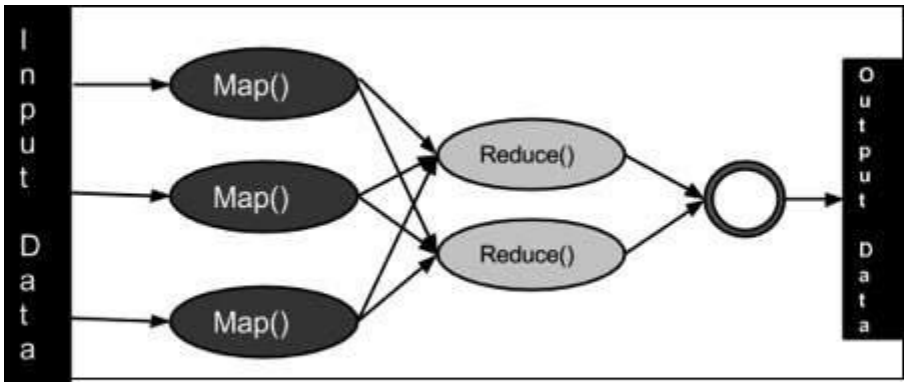
MapReduce

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into *mappers* and *reducers* is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

The Algorithm

- Generally MapReduce paradigm is based on sending the computer to where the data resides.
- MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
    - **Map stage** − The map or mapper’s job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.
    - **Reduce stage** − This stage is the combination of the **Shuffle** stage and the **Reduce** stage. The Reducer’s job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.
- During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.
- The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes.
- Most of the computing takes place on nodes with data on local disks that reduces the network traffic.
- After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.



Inputs and Outputs

The MapReduce framework operates on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

The key and the value classes should be in serialized manner by the framework and hence, need to implement the Writable interface. Additionally, the key classes have to implement the Writable-Comparable interface to facilitate sorting by the framework. Input and Output types of a **MapReduce job** − (Input) <k1, v1> → map → <k2, v2> → reduce → <k3, v3>(Output).

	Input	**Output**
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)