10.1 Remote Procedure Call (RPC)

In distributed computing a *remote procedure call* (**RPC**) *is when a* computer program causes a procedure (subroutine) to execute in another address space *(commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer* explicitly coding the details for the remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. This is a form of client–server interaction (caller is client, executer is server), typically implemented via a request–response message-passing system.

RPCs are a form of inter-process communication (IPC), in that *different processes have different address spaces*: if on the same host machine, they have distinct virtual address spaces, even though the physical address space is the same; while if they are on different hosts, the physical address space is different.

How RPC Works

RPC is a kind of request–response protocol. An RPC is initiated by the *client*, which sends a request message to a known remote *server* to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process. While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution), unless the client sends an asynchronous request to the server, such as an XHTTP call. There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols.

An important difference between remote procedure calls and local calls is that remote calls can fail because of unpredictable network problems. Also, callers generally must deal with such failures without knowing whether the remote procedure was actually invoked.

RPC Process

Stub = piece of code that converts parameters passed between client and server

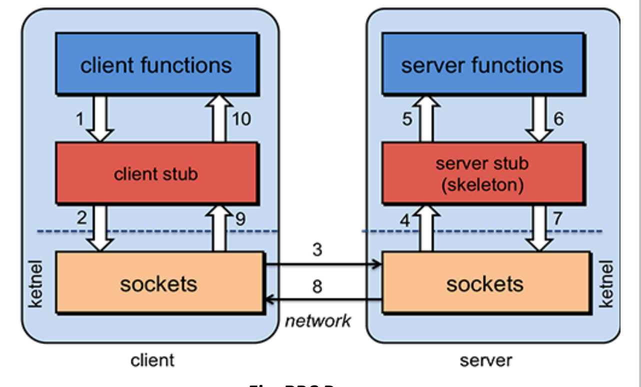


Fig. RPC Process

1. The client calls a local procedure, called the client stub. To the client process, this appears to be the actual procedure, because it is a regular local procedure. It just does something different since the real procedure is on the server. The client stub packages the parameters to the remote procedure (this may involve converting them to a standard format) and builds one or more network messages. The packaging of arguments into a network message is called marshaling and requires serializing all the data elements into a flat array-of-bytes format.
2. Network messages are sent by the client stub to the remote system (via a system call to the local kernel i.e. client OS using *sockets* interfaces).
3. Network messages are transferred by the kernel to the remote system via some protocol (either connectionless or connection-oriented).
4. A server stub, sometimes called the skeleton, receives the messages on the server. It unmarshals the arguments from the messages and, if necessary, converts them from a standard network format into a machine-specific form.
5. The server stub calls the server function (which, to the client, is the remote procedure), passing it the arguments that it received from the client.
6. When the server function is finished, it returns to the server stub with its return values.
7. The server stub converts the return values, if necessary, and marshals them into one or more network messages to send to the client stub.
8. Messages get sent back across the network to the client stub.
9. The client stub reads the messages from the local kernel.
10. The client stub then returns the results to the client function, converting them from the network representation to a local one if necessary.

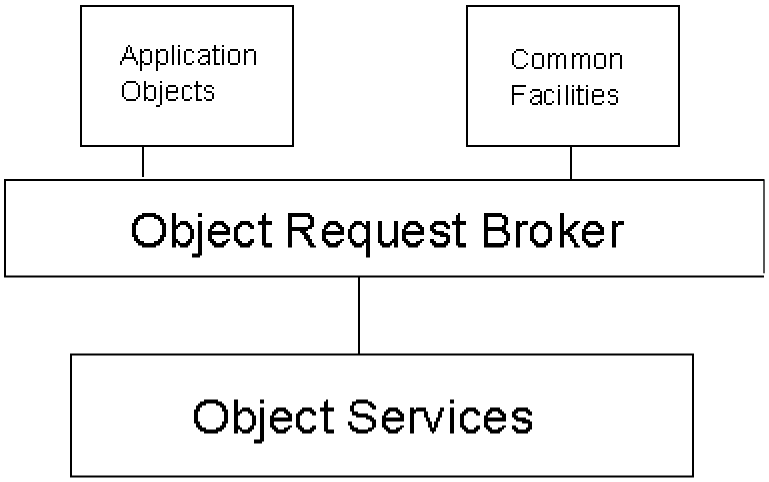
10.2 Object Management Architecture (OMA)

Object Management Group(**OMG**) is an organization that represents over 700 software vendors, software developers, and end users. OMG describes its mission as developing "**The Architecture for a Connected World**." It was established in 1989 to promote object technologies on distributed computing systems. To this end, it has developed a common architectural framework for object-oriented applications based on an open and widely available interface specification called OMA (Object Management Architecture).

OMA is an architecture developed by the OMG (Object Management Group) that provides an industry standard for developing object-oriented applications to run on distributed networks. The goal of the OMG is to provide a common architectural framework for object-oriented applications based on widely available interface specifications.

The OMA reference model identifies and characterizes components, interfaces, and protocols that comprise the OMA. It consists of components that are grouped into application-oriented interfaces, industry-specific vertical applications, object services, and ORBs (object request brokers). The ORB defined by the OMG is known more commonly as CORBA (Common Object Request Broker Architecture).

The OMA is composed of four component categories:



- Object Request Broker (ORB) - It is a communication infrastructure that allows or facilitates object communication. The ORB relays object requests across distributed heterogeneous computing environments.
- Object Services - Object services are low-level system type services (object persistence, transaction capabilities, security, etc..). This collection of system-oriented services allows application developers to construct applications without having to 'reinvent the wheel'. Support for object services must be supported by all ORB environments and platforms.
- Common Facilities - Common facilities are high-level, application-oriented services (such as mail and printing facilities). Unlike Object Services, support for common facilities is discretionary.
- Application Objects - Application objects are developers programs, legacy systems and commercial software. These objects make use of the other three component categories.

10.3 Distributed Resource Architecture

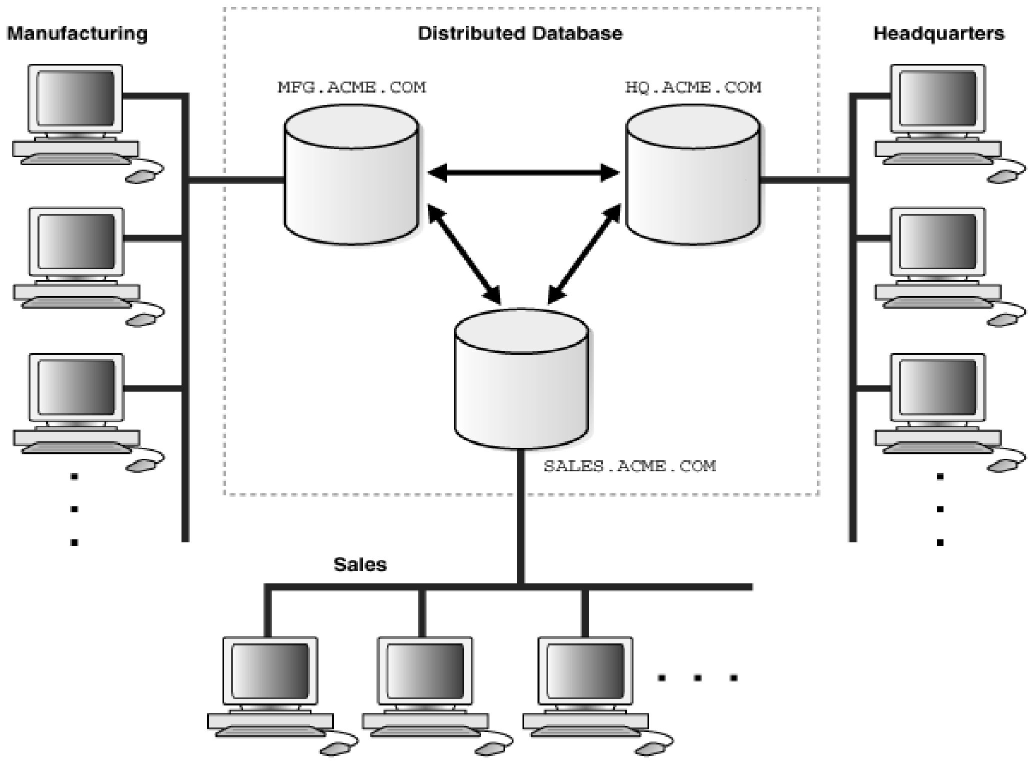
Main goal of a distributed system is to effectively utilize the collective resources of the system, namely, the memory and the processors of the individual nodes. This certain problems affecting to sharing memory and processors in distributed systems.

- Issue arise while sharing memory in a distributed system: memory contention and faults.
- load balancing, which is a mechanism for sharing processors in a distributed system.

Distributed resource system and architecture may be used for generalized computing applications or specialized applications such as graphics and visualization. *The resources can be located remotely from the users because only the results of the execution are transmitted over the network to reduce bandwidth requirements and transmit time*. Therefore, the bulk of the data needed to generate the results remains where they were generated and are accessed on a more frequent basis. *Each resource is operable to access data generally stored co-located therewith, generate the pixel data, and the network compositor is operable to composite the pixel data into one or more graphical images and deliver the images to the dedicated display devices.*

10.3.1 Distributed data Architecture

A distributed database system allows applications to access data from local and remote databases. In a homogenous distributed database system, each database is an Oracle Database. In a heterogeneous distributed database system, at least one of the databases is not an Oracle Database. Distributed databases use a client/server architecture to process information requests.



- Homogenous Distributed Database Systems
- Heterogeneous Distributed Database Systems
- Client/Server Database Architecture

Homogenous Distributed Database Systems

- A homogenous distributed database system is a network of two or more Oracle Databases that reside on one or more machines. Figure 29-1 illustrates a distributed system that connects three databases: hq, mfg, and sales. An application can simultaneously access or modify the data in several databases in a single distributed environment. For example, a single query from a Manufacturing client on local database mfg can retrieve joined data from the products table on the local database and the dept table on the remote hq database.
- For a client application, the location and platform of the databases are transparent. You can also create synonyms for remote objects in the distributed system so that users can access them with the same syntax as local objects. For example, if you are connected to database mfg but want to access data on database hq, creating a synonym on mfg for the remote dept table enables you to issue this query:
- SELECT * FROM dept;

In this way, a distributed system gives the appearance of native data access. Users on mfg do not have to know that the data they access resides on remote databases.

Heterogeneous Distributed Database Systems

- In a heterogeneous distributed database system, at least one of the databases is a non-Oracle Database system. To the application, the heterogeneous distributed database system appears as a single, local, Oracle Database. The local Oracle Database server hides the distribution and heterogeneity of the data.
- The Oracle Database server accesses the non-Oracle Database system using Oracle Heterogeneous Services in conjunction with an agent. If you access the non-Oracle Database data store using an Oracle Transparent Gateway, then the agent is a system-specific application. For example, if you include a Sybase database in an Oracle Database distributed system, then you need to obtain a Sybase-specific transparent gateway so that the Oracle Database in the system can communicate with it.
- Alternatively, you can use generic connectivity to access non-Oracle Database data stores so long as the non-Oracle Database system supports the ODBC or OLE DB protocols.

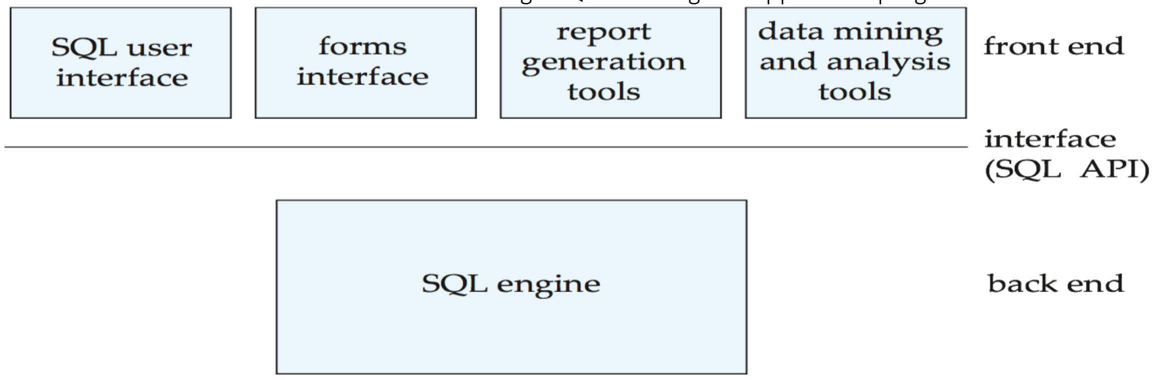
Client/Server Database Architecture - Centralized

A database server managing a database, and a client is an application that requests information from a server. Each computer in a network is a node that can host one or more databases. Each node in a distributed database system can act as a client, a server, or both, depending on the situation.

Database functionality can be divided into:

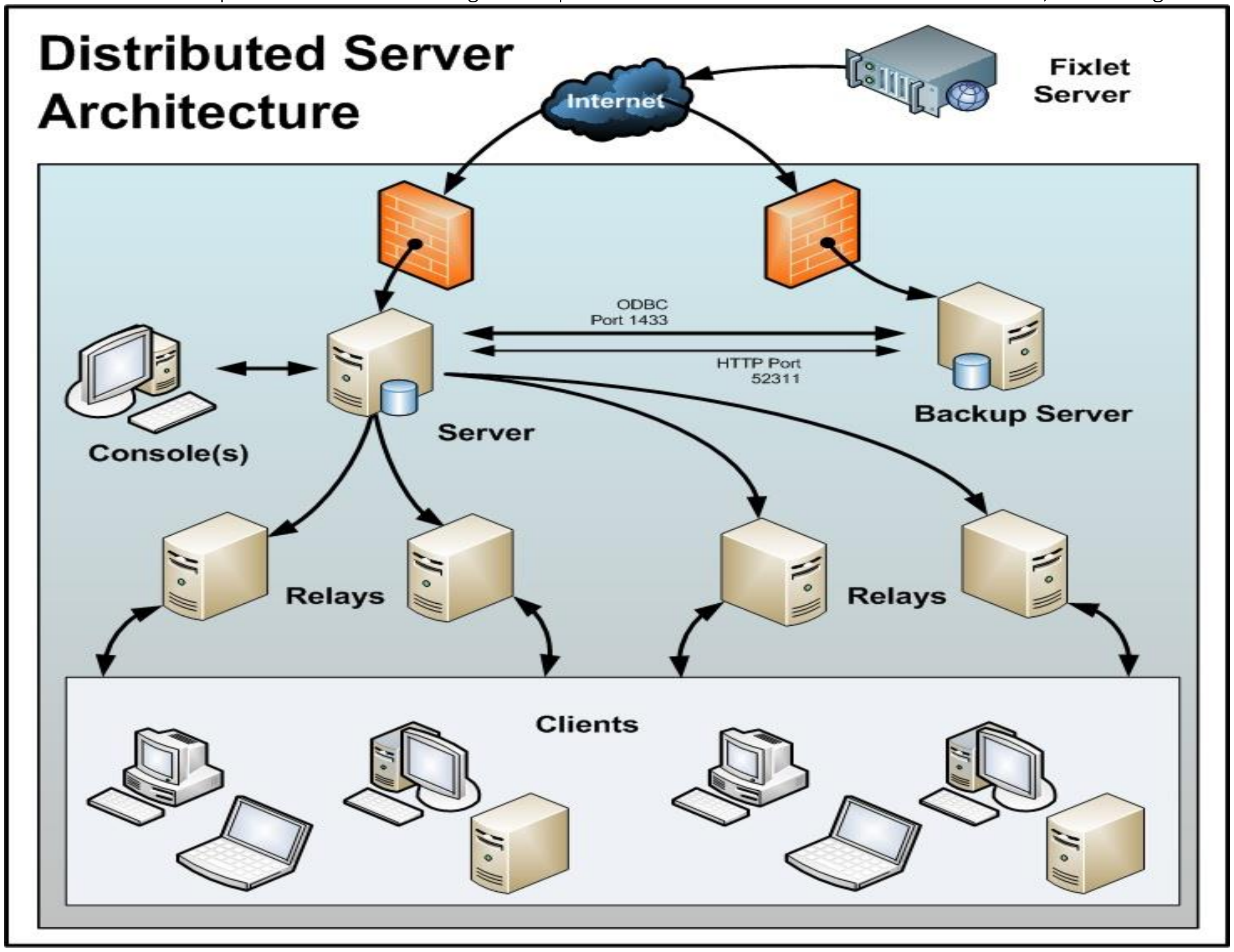
- **Back-end (Server):** manages access structures, query evaluation and optimization, concurrency control and recovery.
- **Front-end (Client)**: consists of tools such as *forms*, *report-writers*, and graphical user interface facilities.

The interface between the front-end and the back-end is through SQL or through an application program interface.



10.3.2 Distributed Server Architecture

The following is a diagram of a typical DSA setup with two servers. Each Server is behind a firewall, possibly in a separate office, although it is easy to set up multiple servers in a single office as well. It is important that the Servers have high-speed connections to replicate the Tivoli Endpoint Manager data (generally LAN speeds of 10-100Mbps are required). The Tivoli Endpoint Manager Servers communicate over ODBC and HTTP protocols. This DSA configuration provides automatic failover and failback services, minimizing loss of data.



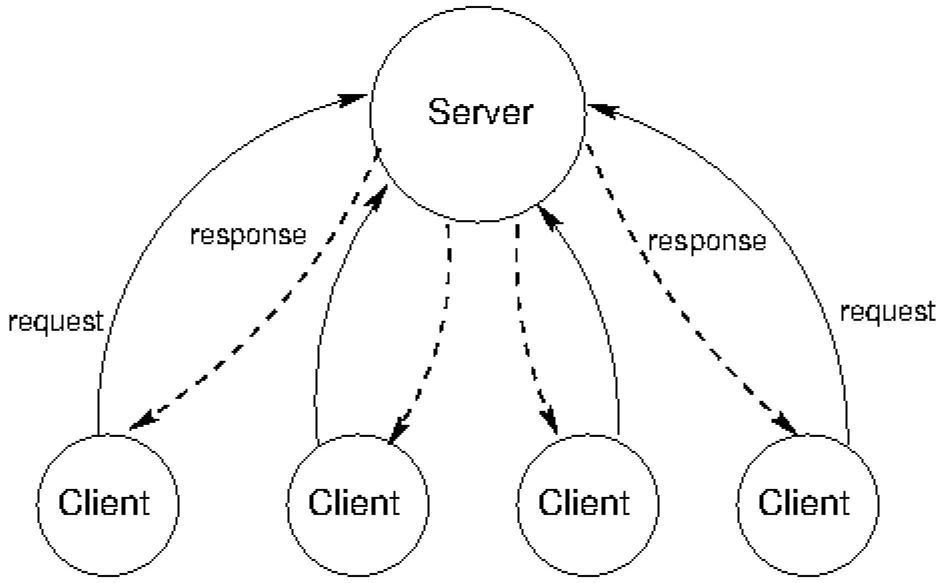
10.3.3 Distributed Computing Architecture

A distributed system is a network of independent computers that communicate with each other in order to achieve a goal. The computers in a distributed system are independent and do not physically share memory or processors. They communicate with each other using *messages*, pieces of information transferred from one computer to another over a network. Messages can communicate many things: computers can tell other computers to execute a procedure with particular arguments, they can send and receive packets of data, or send signals that tell other computers to behave a certain way.

Computers in a distributed system can have different roles. A computer's role depends on the goal of the system and the computer's own hardware and software properties. There are two main ways of organizing computers in a distributed system. The first is the client-server architecture, and the second is the peer-to-peer architecture.

Client/Server Systems

The client-server architecture is a way to dispense a service from a central source. There is a single *server* that provides a service, and multiple *clients* that communicate with the server to consume its products. In this architecture, clients and servers have different jobs. The server's job is to respond to service requests from clients, while a client's job is to use the data provided in response in order to perform some task.



The client-server model of communication can be traced back to the introduction of UNIX in the 1970's, but perhaps the most influential use of the model is the modern World Wide Web. An example of a client-server interaction is reading the New York Times online. When the web server at www.nytimes.com is contacted by a web browsing client (like Firefox), its job is to send back the HTML of the New York Times main page. This could involve calculating personalized content based on user account information sent by the client, and fetching appropriate advertisements. The job of the web browsing client is to render the HTML code sent by the server. This means displaying the images, arranging the content visually, showing different colors, fonts, and shapes and allowing users to interact with the rendered web page.

The concepts of *client* and *server* are powerful functional abstractions. A server is simply a unit that provides a service, possibly to multiple clients simultaneously, and a client is a unit that consumes the service. The clients do not need to know the details of how the service is provided, or how the data they are receiving is stored or calculated, and the server does not need to know how the data is going to be used.

On the web, we think of clients and servers as being on different machines, but even systems on a single machine can have client/server architectures. For example, signals from input devices on a computer need to be generally available to programs running on the computer. The programs are clients, consuming mouse and keyboard input data. The operating system's device drivers are the servers, taking in physical signals and serving them up as usable input.

A drawback of client-server systems is that the server is a single point of failure. It is the only component with the ability to dispense the service. There can be any number of clients, which are interchangeable and can come and go as necessary. If the server goes down, however, the system stops working. Thus, the functional abstraction created by the client-server architecture also makes it vulnerable to failure.

Another drawback of client-server systems is that resources become scarce if there are too many clients. Clients increase the demand on the system without contributing any computing resources. Client-server systems cannot shrink and grow with changing demand.

Peer-to-peer Systems

The client-server model is appropriate for service-oriented situations. However, there are other computational goals for which a more equal division of labor is a better choice. The term *peer-to-peer* is used to describe distributed systems in which labor is divided among all the components of the system. All the computers send and receive data, and they all contribute some processing power and memory. As a distributed system increases in size, its capacity of computational resources increases. In a peer-to-peer system, all components of the system contribute some processing power and memory to a distributed computation.

Division of labor among *all* participants is the identifying characteristic of a peer-to-peer system. This means that peers need to be able to communicate with each other reliably. In order to make sure that messages reach their intended destinations, peer-to-peer systems need to have an organized network structure. The components in these systems cooperate to maintain enough information about the locations of other components to send messages to intended destinations.

In some peer-to-peer systems, the job of maintaining the health of the network is taken on by a set of specialized components. Such systems are not pure peer-to-peer systems, because they have different types of components that serve different functions. The components that support a peer-to-peer network act like scaffolding: they help the network stay connected, they maintain information about the locations of different computers, and they help newcomers take their place within their neighborhood.

The most common applications of peer-to-peer systems are data transfer and data storage. For data transfer, each computer in the system contributes to send data over the network. If the destination computer is in a particular computer's neighborhood, that computer helps send data along. For data storage, the data set may be too large to fit on any single computer, or too valuable to store on just a single computer. Each computer stores a small portion of the data, and there may be multiple copies of the same data spread over different computers. When a computer fails, the data that was on it can be restored from other copies and put back when a replacement arrives.
Skype, the voice- and video-chat service, is an example of a data transfer application with a peer-to-peer architecture. When two people on different computers are having a Skype conversation, their communications are broken up into packets of 1s and 0s and transmitted through a peer-to-peer network. This network is composed of other people whose computers are signed into Skype. Each computer knows the location of a few other computers in its neighborhood. A computer helps send a packet to its destination by passing it on a neighbor, which passes it on to some other neighbor, and so on, until the packet reaches its intended destination. Skype is not a pure peer-to-peer system. A scaffolding network of *supernodes* is responsible for logging-in and logging-out users, maintaining information about the locations of their computers, and modifying the network structure to deal with users entering and leaving.