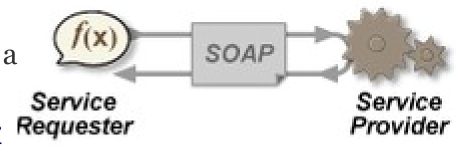
RPC:

Sequence of events

1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
3. The client's local operating system sends the message from the client machine to the server machine.
4. The local operating system on the server machine passes the incoming packets to the server stub.
5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshalling.
6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

XML-RPC is a remote procedure call (RPC) protocol which uses XML to encode its calls and HTTP as a transport mechanism.

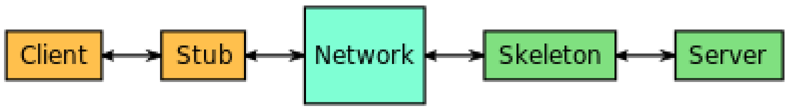
SOAP (successor of XML-RPC):



SOAP (abbreviation for **Simple Object Access Protocol**) is a messaging protocol specification for exchanging structured information in the implementation of web services in computer networks. Its purpose is to provide extensibility, neutrality and independence. It uses XML Information Set for its message format, and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

SOAP allows processes running on disparate operating systems (such as Windows and Linux) to communicate using Extensible Markup Language (XML). Since Web protocols like HTTP are installed and running on all operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms.

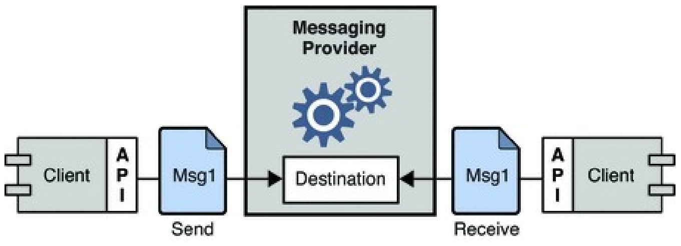
RMI:



In computing, the **Java Remote Method Invocation** (**Java RMI**) is a Java API that performs remote method invocation, the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage-collection.

The original implementation depends on Java Virtual Machine (JVM) class-representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP). In order to support code running in a non-JVM context, programmers later developed a CORBA version.

MOM:



Message-oriented middleware (MOM) is software or hardware infrastructure supporting sending and receiving messages between distributed systems. MOM allows application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols. The middleware creates a distributed communications layer that insulates the application developer from the details of the various operating systems and network interfaces. APIs that extend across diverse platforms and networks are typically provided by MOM.[1]

JMS (a MOM):
The **Java Message Service** (**JMS**) API is a Java message-oriented middleware API for sending messages between two or more clients.[1] It is an implementation to handle the producer–consumer problem. JMS is a part of the Java Platform, Enterprise Edition (Java EE), and was defined by a specification developed at Sun Microsystems, but which has since been guided by the Java Community Process.[2] It is a messaging standard that allows application components based on Java EE to create, send, receive, and read messages. It allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous.[3]

Messaging is a form of *loosely coupled* distributed communication, where in this context the term 'communication' can be understood as an exchange of messages between software components. Message-oriented technologies attempt to relax *tightly coupled* communication (such as TCP network sockets, CORBA or RMI) by the introduction of an intermediary component. This approach allows software components to communicate with each other indirectly. Benefits of this include message senders not needing to have precise knowledge of their receivers.

The advantages of messaging include the ability to integrate heterogeneous platforms, reduce system bottlenecks, increase scalability, and respond more quickly to change.[4]

The JMS API supports two distinct models:

- Point-to-point
- Publish-and-subscribe

COM:

Component Object Model (**COM**) is a binary-interface standard for software components introduced by Microsoft. The essence of COM is a language-neutral way of implementing objects that can be used in environments different from the one in which they were created, even across machine boundaries. For well-authored components, COM allows reuse of objects with no knowledge of their internal implementation, as it forces component implementers to provide well-defined interfaces that are separated from the implementation. For some applications, COM has been replaced at least to some extent by the Microsoft .NET framework, and support for Web Services through the Windows Communication Foundation (WCF). However, COM objects can be used with all .NET languages through .NET COM Interop. Networked DCOM uses binary proprietary formats, while WCF encourages the use of XML-based SOAP messaging. COM is very similar to other component software interface technologies, such as CORBA and Enterprise JavaBeans.

DCE:

In computing, the **Distributed Computing Environment** (**DCE**) software system was developed in the early 1990s from the work of the Open Software Foundation (OSF), a consortium (founded in 1988) that included Apollo Computer (part of Hewlett-Packard from 1989), IBM, Digital Equipment Corporation, and others.[1][2] The DCE supplies a framework and a toolkit for developing client/server applications.[3] The framework includes:

- a remote procedure call (RPC) mechanism known as DCE/RPC
- a naming (directory) service
- a time service
- an authentication service
- a distributed file system (DFS) known as DCE/DFS

DCE represented a big step in the direction of standardization of architectures, which had previously been manufacturer-dependent. Like the OSI model, DCE has not seen much success in practical implementation; however, its underlying concepts have had more substantial influence over subsequent efforts.

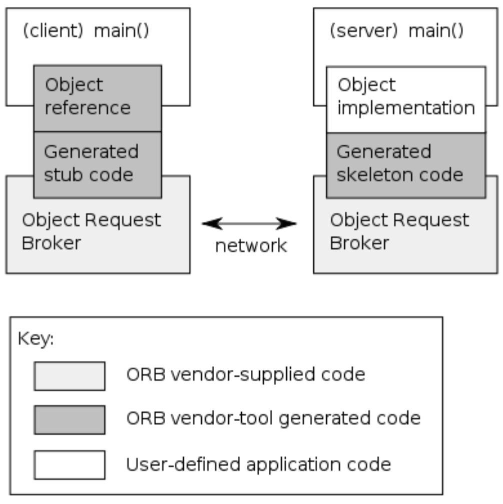
Major components of DCE within every cell are:

1. The **Security Server** that is responsible for authentication
2. The **Cell Directory Server** (CDS) that is the repository of resources and ACLs and
3. The **Distributed Time Server** that provides an accurate clock for proper functioning of the entire cell

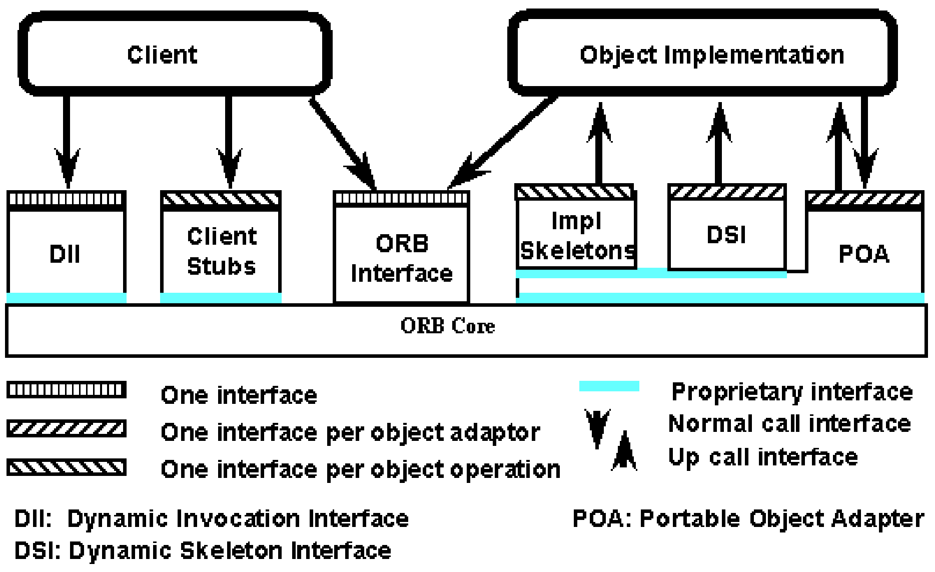
One of the major uses of DCE today is Microsoft's DCOM and ODBC systems, which use DCE/RPC (in MSRPC) as their network transport layer.

CORBA:

The **Common Object Request Broker Architecture** (**CORBA**) is a standard defined by the Object Management Group (OMG) designed to facilitate the communication of systems that are deployed on diverse platforms. CORBA enables collaboration between systems on different operating systems, programming languages, and computing hardware. CORBA uses an object-oriented model although the



systems that use the CORBA do not have to be object-oriented. CORBA is an example of the distributed object paradigm.
CORBA enables communication between software written in different languages and running on different computers. Implementation details from specific operating systems, programming languages, and hardware platforms are all removed from the responsibility of developers who use CORBA.
CORBA uses an interface definition language (IDL) to specify the interfaces that objects present to the outer world. CORBA then specifies a *mapping* from IDL to a specific implementation language like C++ or Java.


Client
Object Implementation
DII
Client Stubs
ORB Interface
Impl Skeletons
DSI
POA
ORB Core
One interface
One interface per object adaptor
One interface per object operation
Proprietary interface
Normal call interface
Up call interface
DII: Dynamic Invocation Interface
DSI: Dynamic Skeleton Interface
POA: Portable Object Adapter

The CORBA specification dictates there shall be an ORB through which an application would interact with other objects. This is how it is implemented in practice:

1. The application simply initializes the ORB, and accesses an internal *Object Adapter*, which maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies.
2. The Object Adapter is used to register instances of the *generated code classes*. Generated code classes are the result of compiling the user IDL code, which translates the high-level interface definition into an OS- and language-specific class base for use by the user application. This step is necessary in order to enforce CORBA semantics and provide a clean user process for interfacing with the CORBA infrastructure.

In addition to providing users with a language and a platform-neutral remote procedure call (RPC) specification, CORBA defines commonly needed services such as transactions and security, events, time, and other domain-specific interface models.

RMI-IIOP is the method that is chosen by Java programmers who want to use the RMI interfaces, but use IIOP as the transport. RMI-IIOP requires that all remote interfaces are defined as Java RMI interfaces. **Java IDL** is an alternative solution, intended for CORBA programmers who want to program in Java to implement objects that are defined in IDL. The general rule that is suggested by Oracle is to use Java IDL when you are using Java to access existing CORBA resources, and RMI-IIOP to export RMI resources to CORBA.

IDL:

An **interface description language** or **interface definition language** (**IDL**), is a specification language used to describe a software component's application programming interface (API). IDLs describe an interface in a language-independent way, enabling communication between software components that do not share one language. For example, between those written in C++ and those written in Java.

IDLs are commonly used in remote procedure call software. In these cases the machines at either end of the *link* may be using different operating systems and computer languages. IDLs offer a bridge between the two different systems.

Software systems based on IDLs include Sun's ONC RPC, The Open Group's Distributed Computing Environment, IBM's System Object Model, the Object Management Group's CORBA (which implements OMG IDL, an IDL based on DCE/RPC) and Data Distribution Service, Mozilla's XPCOM, Microsoft's Microsoft RPC (which evolved into COM and DCOM), Facebook's Thrift and WSDL for Web services.

Enterprise Java Beans: (wikipedia and https://stackify.com/enterprise-java-beans/)

Simply put, an Enterprise Java Bean is a Java class with one or more annotations from the EJB spec which grant the class special powers when running inside of an EJB container.

EJB is one of several Java APIs for modular construction of enterprise software. EJB is a server-side software component that encapsulates business logic of an application. An EJB web container provides a runtime environment for web related software components, including computer security, Java servlet lifecycle management, transaction processing, and other web services. The EJB specification is a subset of the Java EE specification.[1]

The EJB specification provides a standard way to implement the server-side (also called "back-end") 'business' software typically found in enterprise applications (as opposed to 'front-end' user interface software). Such software addresses the same types of problem, and solutions to these problems are often repeatedly re-implemented by programmers. Enterprise JavaBeans is intended to handle such common concerns as persistence, transactional integrity and security in a standard way, leaving programmers free to concentrate on the particular parts of the enterprise software at hand.

- Session beans – invoked by client

A session bean encapsulates business logic that can be invoked programmatically by a client. The invocation can be done locally by another class in the same JVM or remotely over the network from another JVM. The bean performs the task for the client, abstracting its complexity similar to a web service, for example.

```
@Stateless
public class TestStatelessEjb {
   public String sayHello(String name) {
      return "Hello, " + name + "!";
   }
}
```

- Message driven beans - event-driven

A message-driven bean or MDB is an enterprise bean that allows you to process messages asynchronously. This type of bean normally acts as a JMS message listener, which is similar to an event listener but receives JMS messages instead of events.

They are in many ways similar to a Stateless session bean but they are not invoked by a client. instead, they are event-driven

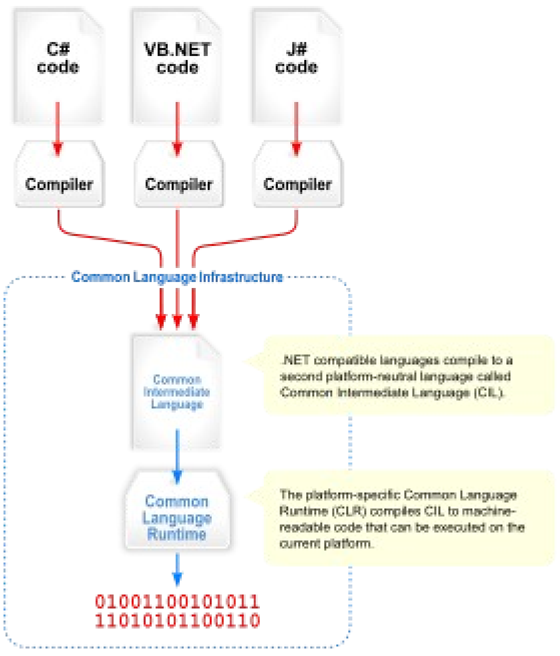
```
@MessageDriven(mappedName = "jms/TestQueue")
public class TestMessageDrivenBean implements MessageListener {

  @Resource
  MessageDrivenContext messageDrivenContext;

  public void onMessage(Message message) {
    try {
      if (message instanceof TextMessage) {
        TextMessage msg = (TextMessage) message;
        msg.getText();
      }
    } catch (JMSException e) {
      messageDrivenContext.setRollbackOnly();
    }
  }
}
```

CLI:

The **Common Language Infrastructure** (**CLI**) is an open specification (technical standard) developed by Microsoft and standardized by ISO[1] and Ecma[2] that describes executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures. This implies it is platform agnostic. The .NET Framework, .NET Core, Mono, DotGNU and Portable.NET are implementations of the CLI.

.NET:

.NET Framework (pronounced as "*dot net*") is a software framework developed by Microsoft that runs primarily on Microsoft Windows. It includes a large class library named as Framework Class Library (FCL) and provides language interoperability (each language can use code written in other languages) across several programming languages. Programs written for .NET Framework execute in a software environment (in contrast to a hardware environment) named the Common Language Runtime (CLR). The CLR is an application virtual machine that provides services such as security, memory management, and exception handling. As such, computer code written using .NET Framework is called "managed code". FCL and CLR together constitute the .NET Framework.


C# code
VB.NET code
J# code
Compiler
Compiler
Compiler
Common Language Infrastructure
Common Intermediate Language
.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).
Common Language Runtime
The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.
01001100101011
11010101100110

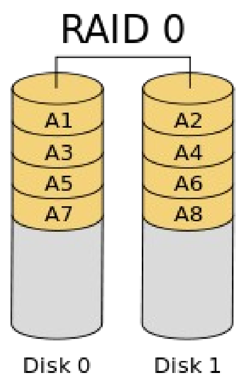
FCL provides user interface, data access, database connectivity, cryptography, web application development, numeric algorithms, and network communications. Programmers produce software by combining their source code with .NET Framework and other libraries.

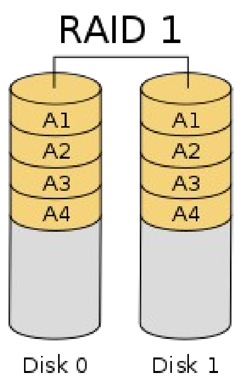
RAID:

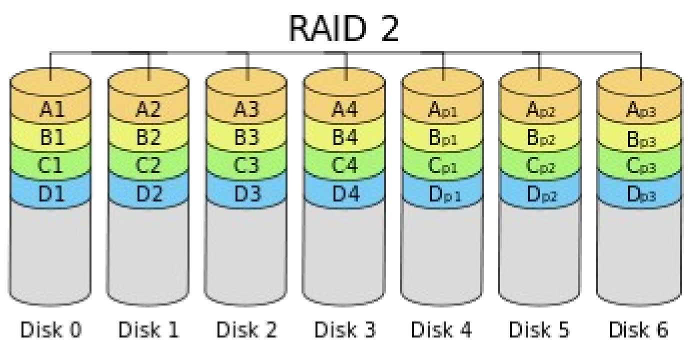
RAID (**Redundant Array of Inexpensive Disks**[1] or Drives, or **Redundant Array of Independent Disks**) is a data storage virtualization technology that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy, performance improvement, or both. This was in contrast to the previous concept of highly reliable mainframe disk drives referred to as "single large expensive disk" (SLED).[2][3]

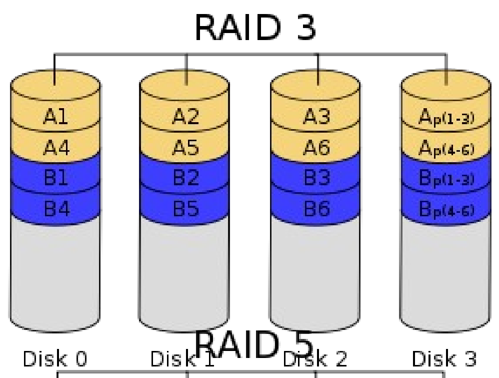
Data is distributed across the drives in one of several ways, referred to as RAID levels, depending on the required level of redundancy and performance. The different schemes, or data distribution layouts, are named by the word "RAID" followed by a number, for example RAID 0 or RAID 1. Each scheme, or RAID level, provides a different balance among the key goals: reliability, availability, performance, and capacity.

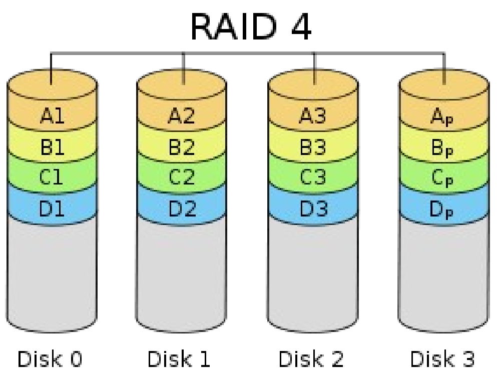
RAID 0 consists of striping, but no mirroring or parity.
RAID 1 consists of data mirroring, without parity or striping.
RAID 2 consists of bit-level striping with dedicated Hamming-code parity.
RAID 3 consists of byte-level striping with dedicated parity.
RAID 4 consists of block-level striping with dedicated parity.
RAID 5 consists of block-level striping with distributed parity.
RAID 6 consists of block-level striping with double distributed parity.

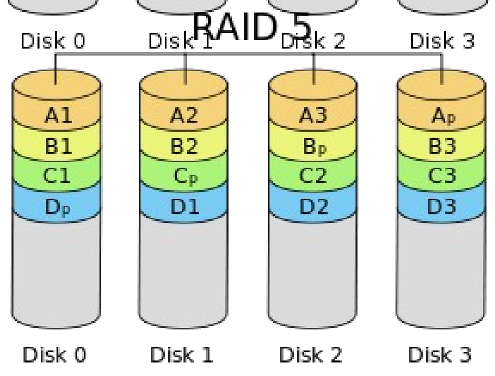












NFS:
is a distributed file system protocol originally developed by Sun Microsystems in 1984,[1] allowing a user on a client computer to access files over a computer network much like local storage is accessed. NFS, like many other protocols, builds on the Open Network Computing Remote Procedure Call (ONC RPC) system. The NFS is an open standard defined in a Request for Comments (RFC), allowing anyone to implement the protocol.

Assuming a Unix-style scenario in which one machine (the client) needs access to data stored on another machine (the NFS server):

1. The server implements NFS daemon processes, running by default as nfsd, to make its data generically available to clients.
2. The server administrator determines what to make available, exporting the names and parameters of directories, typically using the /etc/exports configuration file and the exportfs command.
3. The server security-administration ensures that it can recognize and approve validated clients.
4. The server network configuration ensures that appropriate clients can negotiate with it through any firewall system.
5. The client machine requests access to exported data, typically by issuing a mount command. (The client asks the server (rpcbind) which port the NFS server is using, the client connects to the NFS server (nfsd), nfsd passes the request to mountd)
6. If all goes well, users on the client machine can then view and interact with mounted filesystems on the server within the parameters permitted.

Note that automation of the NFS mounting process may take place — perhaps using /etc/fstab and/or automounting facilities.

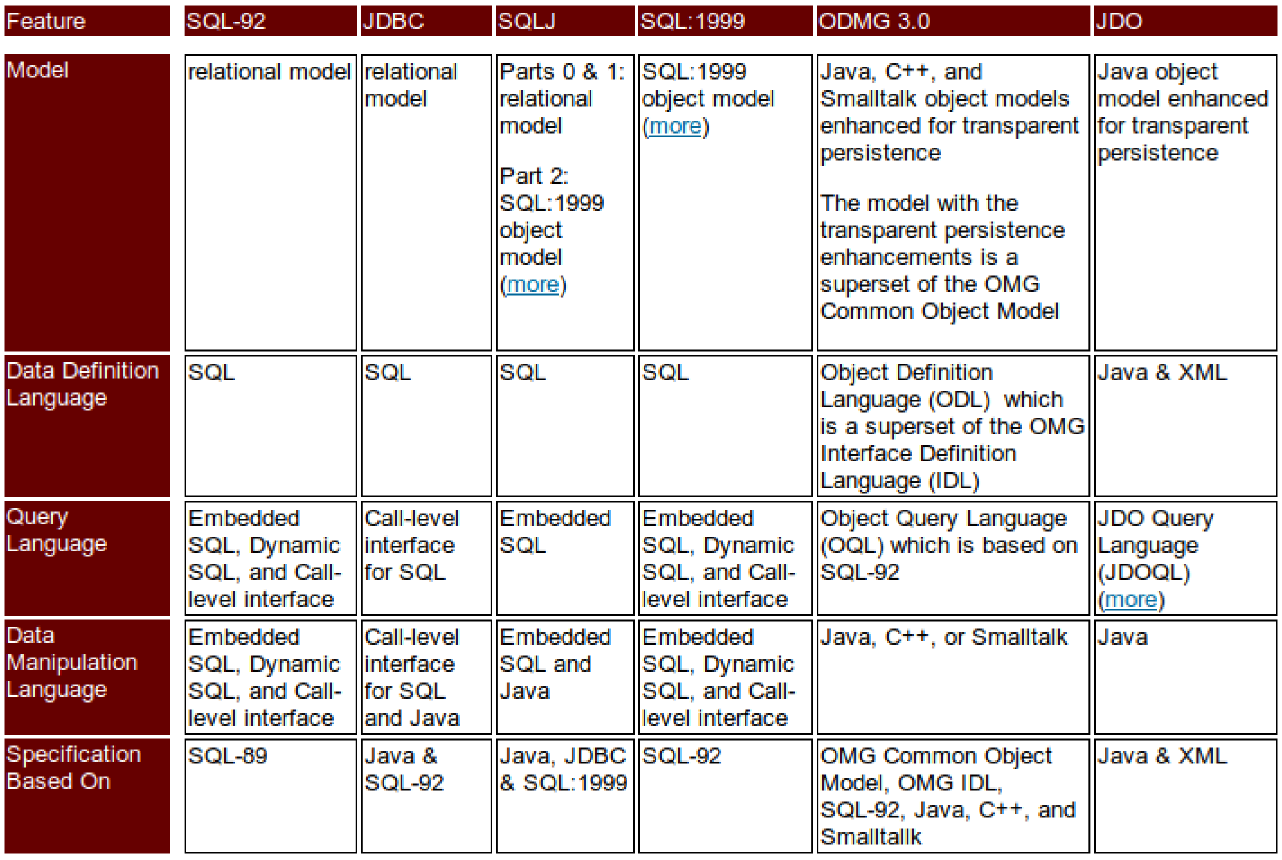
VFS:
A **Virtual File System** (**VFS**) or **virtual filesystem switch** is an abstract layer on top of a more concrete file system. The purpose of a VFS is to allow client applications to access different types of concrete file systems in a uniform way. A VFS can, for example, be used to access local and network storage devices transparently without the client application noticing the difference. It can be used to bridge the differences in Windows, classic Mac OS/macOS and Unix filesystems, so that applications can access files on local file systems of those types without having to know what type of file system they are accessing.

ACID:
In computer science, **ACID** (Atomicity, Consistency, Isolation, Durability) is a set of properties of database transactions intended to guarantee validity even in the event of errors, power failures, etc. In the context of databases, a sequence of database operations that satisfies the ACID properties (and these can be perceived as a single logical operation on the data) is called a transaction.

TP-Monitor:
A **teleprocessing monitor** (also, **Transaction Processing Monitor** or **TP Monitor**) is a control program that monitors the transfer of data between multiple local and remote terminals to ensure that the transaction processes completely or, if an error occurs, to take appropriate actions. [1]

Distributed Transaction Processing:
A **distributed transaction** is a database transaction in which two or more network hosts are involved. Usually, hosts provide **transactional resources**, while the **transaction manager** is responsible for creating and managing a global transaction that encompasses all operations against such resources. Distributed transactions, as any other transactions, must have all four ACID (atomicity, consistency, isolation, durability) properties, where atomicity guarantees all-or-nothing outcomes for the unit of work (operations bundle).

JBDC, SQLJ, JDO, ODMG, EJB QL
(https://www.service-architecture.com/articles/database/comparison_of_dbms_standards.html)



Feature	SQL-92	JDBC	SQLJ	SQL:1999	ODMG 3.0	JDO
Model	relational model	relational model	Parts 0 & 1: relational model Part 2: SQL:1999 object model (more)	SQL:1999 object model (more)	Java, C++, and Smalltalk object models enhanced for transparent persistence The model with the transparent persistence enhancements is a superset of the OMG Common Object Model	Java object model enhanced for transparent persistence
Data Definition Language	SQL	SQL	SQL	SQL	Object Definition Language (ODL) which is a superset of the OMG Interface Definition Language (IDL)	Java & XML
Query Language	Embedded SQL, Dynamic SQL, and Call-level interface	Call-level interface for SQL	Embedded SQL	Embedded SQL, Dynamic SQL, and Call-level interface	Object Query Language (OQL) which is based on SQL-92	JDO Query Language (JDOQL) (more)
Data Manipulation Language	Embedded SQL, Dynamic SQL, and Call-level interface	Call-level interface for SQL and Java	Embedded SQL and Java	Embedded SQL, Dynamic SQL, and Call-level interface	Java, C++, or Smalltalk	Java
Specification Based On	SQL-89	Java & SQL-92	Java, JDBC & SQL:1999	SQL-92	OMG Common Object Model, OMG IDL, SQL-92, Java, C++, and Smalltallk	Java & XML