# SOFTWARE ENVIRONMENTS FOR DISTRIBUTED SYSTEMS AND CLOUDS

This section introduces popular software environments for using distributed and cloud computing systems.

## 1. Service-Oriented Architecture (SOA)

In grids/web services, Java, and CORBA, an entity is, respectively, a service, a Java object, and a CORBA distributed object in a variety of languages. These architectures build on the traditional seven Open Systems Interconnection (OSI) layers that provide the base networking abstractions. On top of this we have a base software environment, which would be .NET or Apache Axis for web services, the Java Virtual Machine for Java, and a broker network for CORBA. On top of this base environment one would build a higher level environment reflecting the special features of the distributed computing environment. This starts with entity interfaces and inter-entity communication, which rebuild the top four OSI layers but at the entity and not the bit level. Figure 1.20 shows the layered architecture for distributed entities used in web services and grid systems.

### 1.1 Layered Architecture for Web Services and Grids

The entity interfaces correspond to the Web Services Description Language (WSDL), Java method, and CORBA interface definition language (IDL) specifications in these example distributed systems. These interfaces are linked with customized, high-level communication systems: SOAP, RMI, and IIOP in the three examples. These communication systems support features including particular message patterns (such as Remote Procedure Call or RPC), fault recovery, and specialized routing. Often, these commu-nication systems

are built on message-oriented middleware (enterprise bus) infrastructure such as Web-Sphere MQ or Java Message Service (JMS) which provide rich functionality and support virtualization of routing, senders, and recipients.

In the case of fault tolerance, the features in the Web Services Reliable Messaging (WSRM) framework mimic the OSI layer capability (as in TCP fault tolerance) modified to match the differ-ent abstractions (such as messages versus packets, virtualized addressing) at the entity levels. Secur-ity is a critical capability that either uses or reimplements the capabilities seen in concepts such as Internet Protocol Security (IPsec) and secure sockets in the OSI layers. Entity communication is supported by higher level services for registries, metadata, and management of the entities discussed in Section 5.4.

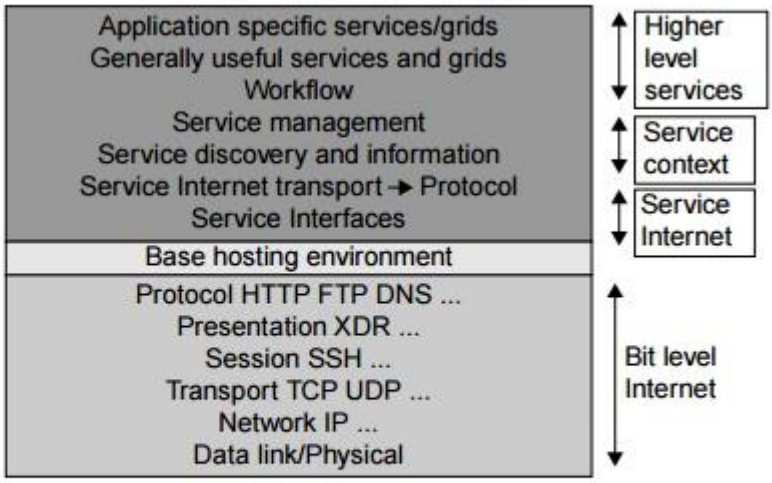


**FIGURE 1.20**

Layered achitecture for web services and the grids.

Here, one might get several models with, for example, JNDI (Jini and Java Naming and Directory Interface) illustrating different approaches within the Java distributed object model. The CORBA Trading Service, UDDI (Universal Description, Discovery, and Integration), LDAP (Lightweight Directory

Access Protocol), and ebXML (Electronic Business using eXtensible Markup Language) are other examples of discovery and information services described in Section 5.4. Management services include service state and lifetime support; examples include the CORBA Life Cycle and Persistent states, the different Enterprise JavaBeans models, Jini's lifetime model, and a suite of web services

specifications in Chapter 5. The above language or interface terms form a collection of entity-level capabilities.

The latter can have performance advantages and offers a “shared memory” model allowing more convenient exchange of information. However, the distributed model has two critical advantages: namely, higher performance (from multiple CPUs when communication is unimportant) and a cleaner separation of software functions with clear software reuse and maintenance advantages. The distributed model is expected to gain popularity as the default approach to software systems. In the earlier years, CORBA and Java approaches were used in distributed systems rather than today’s SOAP, XML, or REST (Representational State Transfer).

## 1.2 Web Services and Tools

Loose coupling and support of heterogeneous implementations make services more attractive than distributed objects. Figure 1.20 corresponds to two choices of service architecture: web services or REST systems (these are further discussed in Chapter 5). Both web services and REST systems have very distinct approaches to building reliable interoperable systems. In web services, one aims to fully specify all aspects of the service and its environment. This specification is carried with communicated messages using Simple Object Access Protocol (SOAP). The hosting environment then becomes a universal distributed operating system with fully distributed capability carried by SOAP messages. This approach has mixed success as it has been hard to agree on key parts of the protocol and even harder to efficiently implement the protocol by software such as Apache Axis.

In the REST approach, one adopts simplicity as the universal principle and delegates most of the difficult problems to application (implementation-specific) software. In a web services language, REST has minimal information in the header, and the message body (that is opaque to generic message processing) carries all the needed information. REST architectures are clearly more appropriate for rapid technology environments. However, the ideas in web services are important and probably will be required in mature systems at a different level in the stack (as part of the application). Note that REST can use XML schemas but not those that are part of SOAP; “XML over HTTP” is a popular design choice in this regard. Above the

communication and management layers, we have the ability to compose new entities or distributed programs by integrating several entities together.

In CORBA and Java, the distributed entities are linked with RPCs, and the simplest way to build composite applications is to view the entities as objects and use the traditional ways of linking them together. For Java, this could be as simple as writing a Java program with method calls replaced by Remote Method Invocation (RMI), while CORBA supports a similar model with a syntax reflecting the C++ style of its entity (object) interfaces. Allowing the term "grid" to refer to a single service or to represent a collection of services, here sensors represent entities that output data (as messages), and grids and clouds represent collections of services that have multiple message-based inputs and outputs.

## 1.3 The Evolution of SOA

As shown in Figure 1.21, service-oriented architecture (SOA) has evolved over the years. SOA applies to building grids, clouds, grids of clouds, clouds of grids, clouds of clouds (also known as interclouds), and systems of systems in general. A large number of sensors provide data-collection services, denoted in the figure as SS (sensor service). A sensor can be a ZigBee device, a Bluetooth device, a WiFi access point, a personal computer, a GPA, or a wireless phone, among other things. Raw data is collected by sensor services. All the SS devices interact with large or small computers, many forms of grids, databases, the compute cloud, the storage cloud, the filter cloud, the discovery cloud, and so on. Filter services ( fs in the figure) are used to eliminate unwanted raw data, in order to respond to specific requests from the web, the grid, or web services.

A collection of filter services forms a filter cloud. We will cover various clouds for compute, storage, filter, and discovery in Chapters 4, 5, and 6, and various grids, P2P networks, and the IoT in Chapters 7, 8, and 9. SOA aims to search for, or sort out, the useful data from the massive

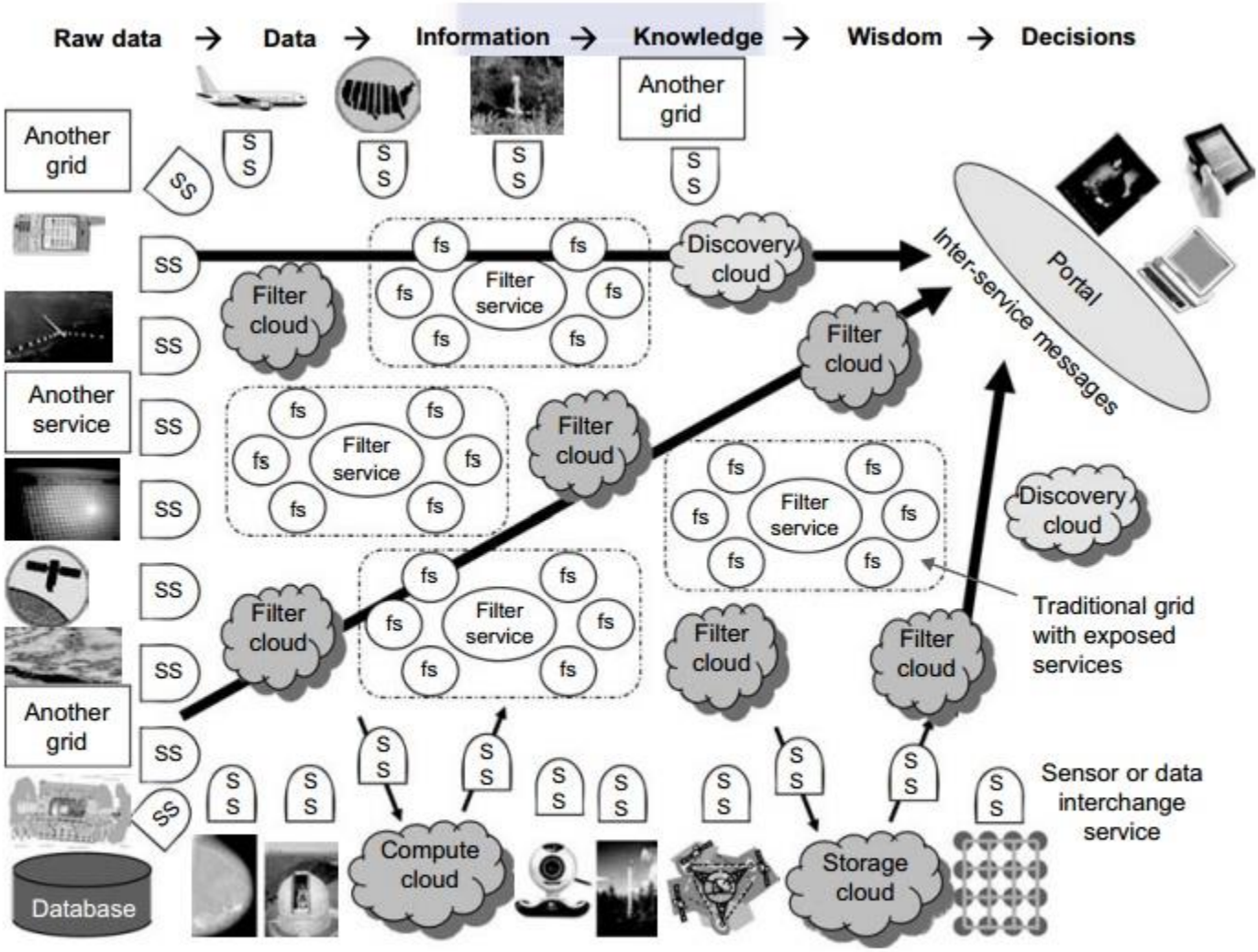


**FIGURE 1.21**

The evolution of SOA: grids of clouds and grids, where "SS" refers to a sensor service and "fs" to a filter or transforming service.

amounts of raw data items. Processing this data will generate useful information, and subsequently, the knowledge for our daily use. In fact, wisdom or intelligence is sorted out of large knowledge bases. Finally, we make intelligent decisions based on both biological and machine wisdom. Read-ers will see these structures more clearly in subsequent chapters.

Most distributed systems require a web interface or portal. For raw data collected by a large number of sensors to be transformed into useful information or knowledge, the data stream may go through a sequence of compute, storage, filter, and discovery clouds. Finally, the inter-service messages converge at the portal,

which is accessed by all users. Two example portals, OGFCE and HUBzero, are described in Section 5.3 using both web service (portlet) and Web 2.0 (gadget) technologies. Many distributed programming models are also built on top of these basic constructs.

### 1.4 Grids versus Clouds

The boundary between grids and clouds are getting blurred in recent years. For web services, work-flow technologies are used to coordinate or orchestrate services with certain specifications used to define critical business process models such as two-phase transactions. Section 5.2 discusses the general approach used in workflow, the BPEL Web Service standard, and several important workflow approaches including Pegasus, Taverna, Kepler, Trident, and Swift. In all approaches, one is building a collection of services which together tackle all or part of a distributed computing problem.

In general, a grid system applies static resources, while a cloud emphasizes elastic resources. For some researchers, the differences between grids and clouds are limited only in dynamic resource allocation based on virtualization and autonomic computing. One can build a grid out of multiple clouds. This type of grid can do a better job than a pure cloud, because it can explicitly support negotiated resource allocation. Thus one may end up building with a system of systems: such as a cloud of clouds, a grid of clouds, or a cloud of grids, or inter-clouds as a basic SOA architecture.

## 2. Trends toward Distributed Operating Systems

The computers in most distributed systems are loosely coupled. Thus, a distributed system inherently has multiple system images. This is mainly due to the fact that all node machines run with an independent operating system. To promote resource sharing and fast communication among node machines, it is best to have a distributed OS that manages all resources coherently and efficiently. Such a system is most likely to be a closed system, and it will likely rely on message passing and RPCs for internode communications. It should be pointed out that a distributed

OS is crucial for upgrading the performance, efficiency, and flexibility of distributed applications.

## 2.1 Distributed Operating Systems

Tanenbaum [26] identifies three approaches for distributing resource management functions in a distributed computer system. The first approach is to build a network OS over a large number of heterogeneous OS platforms. Such an OS offers the lowest transparency to users, and is essentially a distributed file system, with independent computers relying on file sharing as a means of communication. The second approach is to develop middleware to offer a limited degree of resource sharing, similar to the MOSIX/OS developed for clustered systems (see Section 2.4.4). The third approach is to develop a truly distributed OS to achieve higher use or system transparency. Table 1.6 compares the functionalities of these three distributed operating systems.

**Table 1.6** Feature Comparison of Three Distributed Operating Systems

| Distributed OS Functionality | AMOEBA Developed at Vrije University [46] | DCE as OSF/1 by Open Software Foundation [7] | MOSIX for Linux Clusters at Hebrew University [3] |
|---|---|---|---|
| History and Current System Status | Written in C and tested in the European community; version 5.2 released in 1995 | Built as a user extension on top of UNIX, VMS, Windows, OS/2, etc. | Developed since 1977, now called MOSIX2 used in HPC Linux and GPU clusters |
| Distributed OS Architecture | Microkernel-based and location-transparent, uses many servers to handle files, directory, replication, run, boot, and TCP/IP services | Middleware OS providing a platform for running distributed applications; The system supports RPC, security, and threads | A distributed OS with resource discovery, process migration, runtime support, load balancing, flood control, configuration, etc. |
| OS Kernel, Middleware, and Virtualization Support | A special microkernel that handles low-level process, memory, I/O, and communication functions | DCE packages handle file,time, directory, security services, RPC, and authentication at middleware or user space | MOSIX2 runs with Linux 2.6; extensions for use in multiple clusters and clouds with provisioned VMs |
| Communication Mechanisms | Uses a network-layer FLIP protocol and RPC to implement point-to-point and group communication | RPC supports authenticated communication and other security services in user programs | Using PVM, MPI in collective communications, priority process control, and queuing services |

## 3. Parallel and Distributed Programming Models

In this section, we will explore four programming models for distributed computing with expected scalable performance and application flexibility. Table 1.7 summarizes three of these models, along with some software tool sets developed in recent years. As we will discuss, MPI is the most popular programming model for message-passing systems. Google's MapReduce and BigTable are for

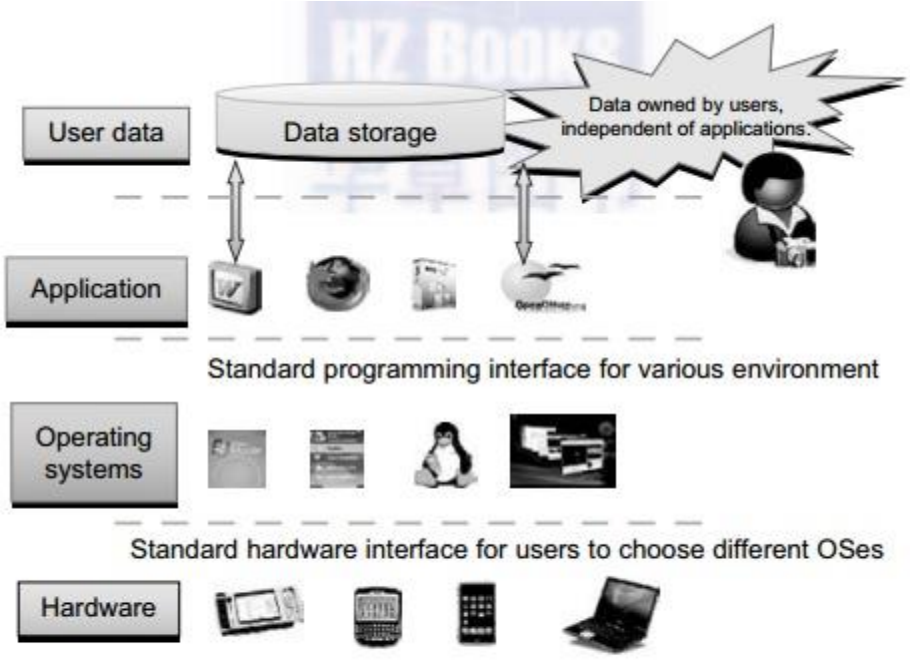


**FIGURE 1.22**

A transparent computing environment that separates the user data, application, OS, and hardware in time and space – an ideal model for cloud computing.

**Table 1.7** Parallel and Distributed Programming Models and Tool Sets

| Model | Description | Features |
|---|---|---|
| MPI | A library of subprograms that can be called from C or FORTRAN to write parallel programs running on distributed computer systems [6,28,42] | Specify synchronous or asynchronous point-to-point and collective communication commands and I/O operations in user programs for message-passing execution |
| MapReduce | A web programming model for scalable data processing on large clusters over large data sets, or in web search operations [16] | *Map* function generates a set of intermediate key/value pairs; *Reduce* function merges all intermediate values with the same key |
| Hadoop | A software library to write and run large user applications on vast data sets in business applications (http://hadoop.apache.org/core) | A scalable, economical, efficient, and reliable tool for providing users with easy access of commercial clusters |

effective use of resources from Internet clouds and data centers. Service clouds demand extending Hadoop, EC2, and S3 to facilitate distributed computing over distributed storage systems. Many other models have also been proposed or developed in the past. In Chapters 5 and 6, we will discuss parallel and distributed programming in more details.

## 3.1 Message-Passing Interface (MPI)

This is the primary programming standard used to develop parallel and concurrent programs to run on a distributed system. MPI is essentially a library of subprograms that can be called from C or FORTRAN to write parallel programs running on a distributed system. The idea is to embody clusters, grid systems, and P2P systems with upgraded web services and utility computing applications. Besides MPI, distributed programming can be also supported with low-level primitives such as the Parallel Virtual Machine (PVM). Both MPI and PVM are described in Hwang and Xu [28].

## 3.2 MapReduce

This is a web programming model for scalable data processing on large clusters over large data sets [16]. The model is applied mainly in web-scale search and cloud

computing applications. The user specifies a Map function to generate a set of intermediate key/value pairs. Then the user applies a Reduce function to merge all intermediate values with the same intermediate key. MapReduce is highly scalable to explore high degrees of parallelism at different job levels. A typical MapReduce computation process can handle terabytes of data on tens of thousands or more client machines. Hundreds of MapReduce programs can be executed simultaneously; in fact, thousands of MapRe-duce jobs are executed on Google's clusters every day.

## 3.3 Hadoop Library

Hadoop offers a software platform that was originally developed by a Yahoo! group. The pack-age enables users to write and run applications over vast amounts of distributed data. Users can easily scale Hadoop to store and process petabytes of data in the web space. Also, Hadoop is economical in that it comes with an open source version of MapReduce that minimizes overhead

**Table 1.8** Grid Standards and Toolkits for Scientific and Engineering Applications [6]

| Standards | Service Functionalities | Key Features and Security Infrastructure |
|---|---|---|
| OGSA Standard | Open Grid Services Architecture; offers common grid service standards for general public use | Supports a heterogeneous distributed environment, bridging CAs, multiple trusted intermediaries, dynamic policies, multiple security mechanisms, etc. |
| Globus Toolkits | Resource allocation, Globus security infrastructure (GSI), and generic security service API | Sign-in multisite authentication with PKI, Kerberos, SSL, Proxy, delegation, and GSS API for message integrity and confidentiality |
| IBM Grid Toolbox | AIX and Linux grids built on top of Globus Toolkit, autonomic computing, replica services | Uses simple CA, grants access, grid service (ReGS), supports grid application for Java (GAF4J), GridMap in IntraGrid for security update |

in task spawning and massive data communication. It is efficient, as it processes data with a high degree of parallelism across a large number of commodity nodes, and it is reliable in that it auto-matically keeps multiple data copies to facilitate redeployment of computing tasks upon unexpected system failures.

## 3.4 Open Grid Services Architecture (OGSA)

The development of grid infrastructure is driven by large-scale distributed computing applications. These applications must count on a high degree of resource and data sharing. Table 1.8 introduces OGSA as a common standard for general public use of grid services. Genesis II is a realization of OGSA. Key features include a distributed execution environment, Public Key Infrastructure (PKI) services using a local certificate authority (CA), trust management, and security policies in grid computing.