Distributed Computing Architecture

Lloyd Field

First Edition, 2012

ISBN 978-81-323-3432-3

Published by:
Research World
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter-1

Computer Architecture

In computer science and computer engineering, **computer architecture** or **digital computer organization** is the conceptual design and fundamental operational structure of a computer system. It's a blueprint and functional description of requirements and design implementations for the various parts of a computer, focusing largely on the way by which the central processing unit (CPU) performs internally and accesses addresses in memory.

It may also be defined as the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.

Computer architecture comprises at least three main subcategories:

- *Instruction set architecture*, or ISA, is the abstract image of a computing system that is seen by a machine language (or assembly language) programmer, including the instruction set, word size, memory address modes, processor registers, and address and data formats.

- *Microarchitecture*, also known as *Computer organization* is a lower level, more concrete and detailed, description of the system that involves how the constituent parts of the system are interconnected and how they interoperate in order to implement the ISA. The size of a computer's cache for instance, is an organizational issue that generally has nothing to do with the ISA.

- *System Design* which includes all of the other hardware components within a computing system such as:

1. System interconnects such as computer buses and switches
2. Memory controllers and hierarchies
3. CPU off-load mechanisms such as direct memory access (DMA)
4. Issues like multiprocessing.

Once both ISA and microarchitecture have been specified, the actual device needs to be designed into hardware. This design process is called the *implementation*.

Implementation is usually not considered architectural definition, but rather hardware design engineering.

Implementation can be further broken down into three (not fully distinct) pieces:

- **Logic Implementation** — design of blocks defined in the microarchitecture at (primarily) the register-transfer and gate levels.
- **Circuit Implementation** — transistor-level design of basic elements (gates, multiplexers, latches etc.) as well as of some larger blocks (ALUs, caches etc.) that may be implemented at this level, or even (partly) at the physical level, for performance reasons.
- **Physical Implementation** — physical circuits are drawn out, the different circuit components are placed in a chip floorplan or on a board and the wires connecting them are routed.

For CPUs, the entire implementation process is often called CPU design.

More specific usages of the term include more general wider-scale hardware architectures, such as cluster computing and Non-Uniform Memory Access (NUMA) architectures.

History

The term “architecture” in computer literature can be traced to the work of Lyle R. Johnson, Muhammad Usman Khan and Frederick P. Brooks, Jr., members in 1959 of the Machine Organization department in IBM’s main research center. Johnson had the opportunity to write a proprietary research communication about Stretch, an IBM-developed supercomputer for Los Alamos Scientific Laboratory. In attempting to characterize his chosen level of detail for discussing the luxuriously embellished computer, he noted that his description of formats, instruction types, hardware parameters, and speed enhancements was at the level of “system architecture” – a term that seemed more useful than “machine organization”. Subsequently, Brooks, one of the Stretch designers, started Chapter 2 of a book (Planning a Computer System: Project Stretch, ed. W. Buchholz, 1962) by writing, “Computer architecture, like other architecture, is the art of determining the needs of the user of a structure and then designing to meet those needs as effectively as possible within economic and technological constraints”. Brooks went on to play a major role in the development of the IBM System/360 line of computers, where “architecture” gained currency as a noun with the definition “what the user needs to know”. Later the computer world would employ the term in many less-explicit ways.

"The way by which the CPU performs internally and accesses memory,” mentioned above, slips into the definition of computer architecture.

Computer architectures

There are many types of computer architectures:

- Quantum computer vs Chemical computer
- Scalar processor vs Vector processor
- Non-Uniform Memory Access (NUMA) computers
- Register machine vs Stack machine
- Harvard architecture vs von Neumann architecture
- Cellular architecture

The quantum computer architecture holds the most promise to revolutionize computing.

Computer architecture topics

Sub-definitions

Some practitioners of computer architecture at companies such as Intel and AMD use more fine distinctions:

- Macroarchitecture — architectural layers that are more abstract than microarchitecture, e.g. ISA
- Instruction Set Architecture (ISA) — as defined above minus
- Assembly ISA — a smart assembler may convert an abstract assembly language common to a group of machines into slightly different machine language for different implementations
- Programmer Visible Macroarchitecture — higher level language tools such as compilers may define a consistent interface or contract to programmers using them, abstracting differences between underlying ISA, UISA, and microarchitectures. E.g. the C, C++, or Java standards define different Programmer Visible Macroarchitecture — although in practice the C microarchitecture for a particular computer includes
- UISA (Microcode Instruction Set Architecture) — a family of machines with different hardware level microarchitectures may share a common microcode architecture, and hence a UISA.
- Pin Architecture — the set of functions that a microprocessor is expected to provide, from the point of view of a hardware platform. E.g. the x86 A20M, FERR/IGNNE or FLUSH pins, and the messages that the processor is expected to emit after completing a cache invalidation so that external caches can be invalidated. Pin architecture functions are more flexible than ISA functions - external hardware can adapt to changing encodings, or changing from a pin to a message - but the functions are expected to be provided in successive implementations even if the manner of encoding them changes.

The role of computer architecture

Computer architecture: the definition

The coordination of abstract levels of a processor under changing forces, involving design, measurement and evaluation. It also includes the overall fundamental working principle of the internal logical structure of a computer system.

It can also be defined as the design of the task-performing part of computers, i.e. how various gates and transistors are interconnected and are caused to function per the instructions given by an assembly language programmer.

Instruction set architecture

1. The ISA is the interface between the software and hardware.
2. It is the set of instructions that bridges the gap between high level languages and the hardware.
3. For a processor to understand a command, it should be in binary and not in High Level Language. The ISA encodes these values.
4. The ISA also defines the items in the computer that are available to a programmer. For example, it defines data types, registers, addressing modes, memory organization etc.
5. Register are high Addressing modes are the ways in which the instructions locate their operands.

Memory organization defines how instructions interact with the memory.

Computer organization

Computer organization helps optimize performance-based products. For example, software engineers need to know the processing ability of processors. They may need to optimize software in order to gain the most performance at the least expense. This can require quite detailed analysis of the computer organization. For example, in a multimedia decoder, the designers might need to arrange for most data to be processed in the fastest data path and the various components are assumed to be in place and task is to investigate the organisational structure to verify the computer parts operates.

Computer organization also helps plan the selection of a processor for a particular project. Multimedia projects may need very rapid data access, while supervisory software may need fast interrupts.

Sometimes certain tasks need additional components as well. For example, a computer capable of virtualization needs virtual memory hardware so that the memory of different simulated computers can be kept separated.

The computer organization and features also affect the power consumption and the cost of the processor.

Design goals

Performance

Computer performance is often described in terms of clock speed (usually in MHz or GHz). This refers to the cycles per second of the main clock of the CPU. However, this metric is somewhat misleading, as a machine with a higher clock rate may not necessarily have higher performance. As a result manufacturers have moved away from clock speed as a measure of performance.

Computer performance can also be measured with the amount of cache a processor has. If the speed, MHz or GHz, were to be a car then the cache is like the gas tank. No matter how fast the car goes, it will still need to get gas. The higher the speed, and the greater the cache, the faster a processor runs.

Modern CPUs can execute multiple instructions per clock cycle, which dramatically speeds up a program. Other factors influence speed, such as the mix of functional units, bus speeds, available memory, and the type and order of instructions in the programs being run.

There are two main types of speed, latency and throughput. Latency is the time between the start of a process and its completion. Throughput is the amount of work done per unit time. Interrupt latency is the guaranteed maximum response time of the system to an electronic event (*e.g.* when the disk drive finishes moving some data). Performance is affected by a very wide range of design choices — for example, pipelining a processor usually makes latency worse (slower) but makes throughput better. Computers that control machinery usually need low interrupt latencies. These computers operate in a real-time environment and fail if an operation is not completed in a specified amount of time. For example, computer-controlled anti-lock brakes must begin braking almost immediately after they have been instructed to brake.

The performance of a computer can be measured using other metrics, depending upon its application domain. A system may be CPU bound (as in numerical calculation), I/O bound (as in a webserving application) or memory bound (as in video editing). Power consumption has become important in servers and portable devices like laptops.

Benchmarking tries to take all these factors into account by measuring the time a computer takes to run through a series of test programs. Although benchmarking shows strengths, it may not help one to choose a computer. Often the measured machines split on different measures. For example, one system might handle scientific applications quickly, while another might play popular video games more smoothly. Furthermore, designers have been known to add special features to their products, whether in hardware or software, which permit a specific benchmark to execute quickly but which do not offer similar advantages to other, more general tasks.

Power consumption

Power consumption is another design criterion that factors in the design of modern computers. Power efficiency can often be traded for performance or cost benefits. With the increasing power density of modern circuits as the number of transistors per chip scales (Moore's law), power efficiency has increased in importance. Recent processor designs such as the Intel Core 2 put more emphasis on increasing power efficiency. Also, in the world of embedded computing, power efficiency has long been and remains the primary design goal next to performance.

Chapter-2

Software Framework and Network Architecture

Software framework

In computer programming, a **software framework** is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code, thus providing specific functionality. Frameworks are a special case of software libraries in that they are reusable abstractions of code wrapped in a well-defined Application programming interface (API), yet they contain some key distinguishing features that separate them from normal libraries.

Software frameworks have these distinguishing features that separate them from libraries or normal user applications:

1. **inversion of control** - In a framework, unlike in libraries or normal user applications, the overall program's flow of control is not dictated by the caller, but by the framework.
2. **default behavior** - A framework has a default behavior. This default behavior must actually be some useful behavior and not a series of no-ops.
3. **extensibility** - A framework can be extended by the user usually by selective overriding or specialized by user code providing specific functionality.
4. **non-modifiable framework code** - The framework code, in general, is not allowed to be modified. Users can extend the framework, but not modify its code.

There are different types of software frameworks: conceptual, application, domain, platform, component, service, development, etc.

Rationale

The designers of software frameworks aim to facilitate software development by allowing designers and programmers to devote their time to meeting software requirements rather than dealing with the more standard low-level details of providing a working system, thereby reducing overall development time. For example, a team using a web application framework to develop a banking web-site can focus on the operations of account withdrawals rather than the mechanics of request handling and state management.

By way of contrast, an in-house or purpose-built framework might be specified for the same project by a programming team as they begin working the overall job — specifying software needs based on first defining data types, structures and processing has long been taught as a successful strategy for top down design. Contrasting software data, its manipulation, and how a software system's various grades and kinds of users will need to either input, treat, or output the data are then used to specify the user interface(s) — some types of access being privileged and locked to other user types — all defining the overall user interfaces which to the users are the visible in-house Framework for the custom coded project. In such a case, each sort of operation, user interface code and so forth need written and separately integrated into the job at hand also more or less adding to necessary testing and validation.

It can be argued that frameworks add to "code bloat", and that due to customer-demand driven applications needs both competing and complementary frameworks sometimes end up in a product. Further, some cognoscenti argue, because of the complexity of their APIs, the intended reduction in overall development time may not be achieved due to the need to spend additional time learning to use the framework — which criticism is certainly valid when a special or new framework is first encountered by a development staff. If such a framework is not utilized in subsequent job taskings, the time invested ascending the framework's learning curve might be more expensive than purpose written code familiar to the project's staff — many programmers keep aside useful boilerplate for common needs.

- However, it could be argued that once the framework is learned, future projects might be quicker and easier to complete — the concept of a framework is to make a one-size-fits-all solution set, and with familiarity, code production should logically be increased. There are no such claims made about the size of the code eventually bundled with the output product, nor its relative efficiency and conciseness. Using any library solution necessarily pulls in extras and unused extraneous assets unless the software is a compiler-object linker making a tight (small, wholly controlled, and specified) executable module.

- The issue continues, but a decade-plus of industry experience has shown that the most effective frameworks turn out to be those that evolve from re-factoring the common code of the enterprise, as opposed to using a generic "one-size-fits-all" framework developed by third-parties for general purposes. An example of that would be how the user interface in such an application package as an Office suite grows to have common look, see, feel and data sharing attributes and methods as the once disparate bundled applications become unified — hopefully a suite which is tighter and smaller as the newer evolved one can be a product sharing integral utility libraries and user interfaces.

This trend in the controversy brings up an important issue about frameworks. Creating a framework that is elegant, as opposed to one that merely solves a problem, is still an art rather than a science. "Software elegance" implies clarity, conciseness, and little waste (extra or extraneous functionality — much of which is user defined). For those frameworks that generate code, for example, "elegance" would imply the creation of code that is clean and comprehensible to a reasonably knowledgeable programmer (and which is therefore readily modifiable), as opposed to one that merely generates correct code. The elegance issue is why relatively few software frameworks have stood the test of time: the best frameworks have been able to evolve gracefully as the underlying technology on which they were built advanced. Even there, having evolved, many such packages will retain legacy capabilities bloating the final software as otherwise replaced methods have been retained in parallel with the newer methods.

Examples

Software frameworks typically contain considerable housekeeping and utility code in order to help bootstrap user applications, but generally focus on specific problem domains, such as:

- Artistic drawing, music composition, and mechanical CAD
- Compilers for different programming languages and target machines.
- Financial modeling applications
- Earth system modeling applications
- Decision support systems
- Media playback and authoring
- Web applications
- Middleware

Architecture

According to Pree, software frameworks consist of *frozen spots* and *hot spots*. *Frozen spots* define the overall architecture of a software system, that is to say its basic components and the relationships between them. These remain unchanged (frozen) in any instantiation of the application framework. *Hot spots* represent those parts where the programmers using the framework add their own code to add the functionality specific to their own project.

Software frameworks define the places in the architecture where application programmers may make adaptations for specific functionality—the hot spots.

In an object-oriented environment, a framework consists of abstract and concrete classes. Instantiation of such a framework consists of composing and subclassing the existing classes.

When developing a concrete software system with a software framework, developers utilize the hot spots according to the specific needs and requirements of the system. Software frameworks rely on the Hollywood Principle: "Don't call us, we'll call you." This means that the user-defined classes (for example, new subclasses), receive messages from the predefined framework classes. Developers usually handle this by implementing superclass abstract methods.

Network architecture

Network architecture is the design of a communications network. It is a framework for the specification of a network's physical components and their functional organization and configuration, its operational principles and procedures, as well as data formats used in its operation.

In telecommunication, the specification of a network architecture may also include a detailed description of products and services delivered via a communications network, as well as detailed rate and billing structures under which services are compensated.

The network architecture of the Internet is predominantly expressed by its use of the Internet Protocol Suite, rather than a specific model for interconnecting networks or nodes in the network, or the usage of specific types of hardware links.

OSI Network Model

The Open Systems Interconnection model (OSI model) is a product of the Open Systems Interconnection effort at the International Organization for Standardization. It is a way of sub-dividing a communications system into smaller parts called layers. A layer is a collection of similar functions that provide services to the layer above it and receives services from the layer below it. On each layer, an instance provides services to the instances at the layer above and requests service from the layer below.

Physical Layer

The Physical Layer defines the electrical and physical specifications for devices. In particular, it defines the relationship between a device and a transmission medium, such as a copper or optical cable. This includes the layout of pins, voltages, cable specifications, hubs, repeaters, network adapters, host bus adapters (HBA used in storage area networks) and more. Its main task is the transmission of a stream of bits over a communication channel.

Data Linking Layer

The Data Link Layer provides the functional and procedural means to transfer data between network entities and to detect and possibly correct errors that may occur in the Physical Layer. Originally, this layer was intended for point-to-point and point-to-multipoint media, characteristic of wide area media in the telephone system. Local area network architecture, which included broadcast-capable multiaccess media, was developed independently of the ISO work in IEEE Project 802. IEEE work assumed sublayering and management functions not required for WAN use. In modern practice, only error detection, not flow control using sliding window, is present in data link protocols such as Point-to-Point Protocol (PPP), and, on local area networks, the IEEE 802.2 LLC layer is not used for most protocols on the Ethernet, and on other local area networks, its flow control and acknowledgment mechanisms are rarely used. Sliding window flow control and acknowledgment is used at the Transport Layer by protocols such as TCP, but is still used in niches where X.25 offers performance advantages. Simply it's main job is to create and recognize the frame boundary this can be done by attaching a special bit patterns to the beginning and the end of the frame ,the input data is broken up into frames

Network Layer

The Network Layer provides the functional and procedural means of transferring variable length data sequences from a source host on one network to a destination host on a different network, while maintaining the quality of service requested by the Transport Layer (in contrast to the data link layer which connects hosts within the same network). The Network Layer performs network routing functions, and might also perform fragmentation and reassembly, and report delivery errors. Routers operate at this layer—sending data throughout the extended network and making the Internet possible. This is a logical addressing scheme – values are chosen by the network engineer. The addressing scheme is not hierarchical.It controls the operation of the subnet and determine the routing strategies between IMP and insures that all the packs are correctly received at the destination in the proper order.

Transport Layer

The Transport Layer provides transparent transfer of data between end users, providing reliable data transfer services to the upper layers. The Transport Layer controls the reliability of a given link through flow control, segmentation/desegmentation, and error control. Some protocols are state and connection oriented. This means that the Transport Layer can keep track of the segments and retransmit those that fail. The Transport layer also provides the acknowledgement of the successful data transmission and sends the next data if no errors occurred. Some Transport Layer protocols, for example TCP, but not UDP, support virtual circuits provideconnection oriented communication over an underlying packet oriented datagram network .Where it assures the delivery of packets in the order in which they were sent and assure that they are free of errors .The datagram transportation deliver the packets randomly and broadcast it to multiple nodes. Notes: The transport layer multiplexes several streams on to 1 physical channel.The transport headers tells which message bongs to which connnection.

The Session Layer

This Layer provide a user interface to the network where the user negotiate to establish a connection ,the user must provide the remote address in with he want to contact. The operation of setting up a session between 2 process is called "Binding" in some protocols it is merged with the transport layer.

Presentation Layer

The Presentation Layer establishes context between Application Layer entities, in which the higher-layer entities may use different syntax and semantics if the presentation service provides a mapping between them. If a mapping is available, presentation service data units are encapsulated into session protocol data units, and passed down the stack.This layer provides independence from data representation (e.g., encryption) by translating between application and network formats. The presentation layer transforms data into the form that the application accepts. This layer formats and encrypts data to be sent across a network. It is sometimes called the syntax layer.The original presentation structure used the basic encoding rules of Abstract Syntax Notation One (ASN.1), with capabilities such as converting an EBCDIC-coded text file to an ASCII-coded file, or serialization of objects and other data structures from and to XML.

Application Layer

The Application Layer is the OSI layer closest to the end user, which means that both the OSI application layer and the user interact directly with the software application. This layer interacts with software applications that implement a communicating component. Such application programs fall outside the scope of the OSI model. Application layer functions typically include identifying communication partners, determining resource availability, and synchronizing communication. When identifying communication partners, the application layer determines the identity and availability of communication partners for an application with data to transmit.

Distributed computing

In distinct usage in distributed computing, the term *network architecture* often describes the structure and classification of a distributed application architecture, as the participating nodes in a distributed application are often referred to as a *network*. For example, the applications architecture of the public switched telephone network (PSTN) has been termed the Advanced Intelligent Network. There are any number of specific classifications but all lie on a continuum between the dumb network (e.g., Internet) and the intelligent computer network (e.g., the telephone network). Other networks contain various elements of these two classical types to make them suitable for various types of applications. Recently the context aware network, which is a synthesis of two, has gained much interest with its ability to combine the best elements of both.

A popular example of such usage of the term in distributed applications, as well as PVCs (permanent virtual circuits), is the organization of nodes in peer-to-peer (P2P) services and networks. P2P networks usually implement overlay networks running over an underlying physical or logical network. These overlay network may implement certain organizational structures of the nodes according to several distinct models, the network architecture of the system.

Network architecture is a broad plan that specifies everything necessary for two application programs on different networks on an Internet to be able to work together effectively.

Chapter-3

Software Architecture

The **software architecture** of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. The term also refers to documentation of a system's software architecture. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects.

Overview

The field of computer science has come across problems associated with complexity since its formation. Earlier problems of complexity were solved by developers by choosing the right data structures, developing algorithms, and by applying the concept of separation of concerns. Although the term "software architecture" is relatively new to the industry, the fundamental principles of the field have been applied sporadically by software engineering pioneers since the mid 1980s. Early attempts to capture and explain software architecture of a system were imprecise and disorganized, often characterized by a set of box-and-line diagrams. During the 1990s there was a concentrated effort to define and codify fundamental aspects of the discipline. Initial sets of design patterns, styles, best practices, description languages, and formal logic were developed during that time.

The software architecture discipline is centered on the idea of reducing complexity through abstraction and separation of concerns. To date there is still no agreement on the precise definition of the term "software architecture".

As a maturing discipline with no clear rules on the right way to build a system, designing software architecture is still a mix of art and science. The "art" aspect of software architecture is because a commercial software system supports some aspect of a business or a mission. How a system supports key business drivers is described via scenarios as non-functional requirements of a system, also known as quality attributes, determine how a system will behave. Every system is unique due to the nature of the business drivers it supports, as such the degree of quality attributes exhibited by a system such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and such other –ilities will vary with each implementation. To bring a software architecture user's perspective into the software architecture, it can be said that software architecture gives the direction to take steps and do the tasks involved in each such user's speciality area and interest e.g. the stakeholders of software systems, the software developer, the software system operational support group, the software maintenance specialists, the deployer, the tester and also the business end user. In this sense software architecture is really the amalgamation of the multiple perspectives a system always embodies. The fact that those several different perspectives can be put together into a software architecture stands as the vindication of the need and justification of creation of software architecture before the software development in a project attains maturity.

History

The origin of software architecture as a concept was first identified in the research work of Edsger Dijkstra in 1968 and David Parnas in the early 1970s. These scientists emphasized that the structure of a software system matters and getting the structure right is critical. The study of the field increased in popularity since the early 1990s with research work concentrating on architectural styles (patterns), architecture description languages, architecture documentation, and formal methods.

Research institutions have played a prominent role in furthering software architecture as a discipline. Mary Shaw and David Garlan of Carnegie Mellon wrote a book titled *Software Architecture: Perspectives on an Emerging Discipline* in 1996, which brought forward the concepts in Software Architecture, such as components, connectors, styles and so on. The University of California, Irvine's Institute for Software Research's efforts in software architecture research is directed primarily in architectural styles, architecture description languages, and dynamic architectures.

The IEEE 1471: ANSI/IEEE 1471-2000: Recommended Practice for Architecture Description of Software-Intensive Systems is the first formal standard in the area of software architecture, and was adopted in 2007 by ISO as *ISO/IEC 42010:2007.*

Software architecture topics

Architecture description languages

Architecture description languages (**ADLs**) are used to describe a Software Architecture. Several different ADLs have been developed by different organizations, including AADL (SAE standard), Wright (developed by Carnegie Mellon), Acme (developed by Carnegie Mellon), xADL (developed by UCI), Darwin (developed by Imperial College London), DAOP-ADL (developed by University of Málaga), and ByADL (University of L'Aquila, Italy). Common elements of an ADL are component, connector and configuration.

Views

Software architecture is commonly organized in views, which are analogous to the different types of blueprints made in building architecture. A view is a representation of a set of system components and relationships among them. Within the ontology established by ANSI/IEEE 1471-2000, *views* are responses to *viewpoints*, where a viewpoint is a specification that describes the architecture in question from the perspective of a given set of stakeholders and their concerns. The viewpoint specifies not only the concerns addressed but the presentation, model kinds used, conventions used and any consistency (correspondence) rules to keep a view consistent with other views.

Some possible views (actually, *viewpoints* in the 1471 ontology) are:

- Functional/logic view
- Code/module view
- Development/structural view
- Concurrency/process/runtime/thread view
- Physical/deployment/install view
- User action/feedback view
- Data view/data model

Several languages for describing software architectures ('architecture description language' in ISO/IEC 42010 / IEEE-1471 terminology) have been devised, but no consensus exists on which symbol-set or language should be used to describe each architecture view. The UML is a standard that can be used *"for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes."* Thus, the UML is a visual language that can be used to create software architecture views.

Architecture frameworks

Frameworks related to the domain of software architecture are:

- 4+1
- RM-ODP (Reference Model of Open Distributed Processing)
- Service-Oriented Modeling Framework (SOMF)

Other architectures such as the Zachman Framework, DODAF, and TOGAF relate to the field of Enterprise architecture.

The distinction from functional design

The IEEE Std 610.12-1990 Standard Glossary of Software Engineering Terminology defines the following distinctions:

- Architectural Design: the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.
- Detailed Design: the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to begin implementation.
- Functional Design: the process of defining the working relationships among the components of a system.
- Preliminary Design: the process of analyzing design alternatives and defining the architecture, components, interfaces, and timing/sizing estimates for a system or components.

Software architecture, also described as strategic design, is an activity concerned with global requirements governing *how* a solution is implemented such as programming paradigms, architectural styles, component-based software engineering standards, architectural patterns, security, scale, integration, and law-governed regularities. Functional design, also described as tactical design, is an activity concerned with local requirements governing *what* a solution does such as algorithms, design patterns, programming idioms, refactorings, and low-level implemenation.

According to the Intension/Locality Hypothesis, the distinction between architectural and detailed design is defined by the Locality Criterion, according to which a statement about software design is non-local (architectural) if and only if a program that satisfies it can be expanded into a program which does not. For example, the client–server style is architectural (strategic) because a program that is built on this principle can be expanded into a program which is not client–server; for example, by adding peer-to-peer nodes.

Architecture is design but not all design is architectural. In practice, the architect is the one who draws the line between software architecture (architectural design) and detailed design (non-architectural design). There aren't rules or guidelines that fit all cases. Examples of rules or heuristics that architects (or organizations) can establish when they want to distinguish between architecture and detailed design include:

- Architecture is driven by non-functional requirements, while functional design is driven by functional requirements.
- Pseudo-code belongs in the detailed design document.
- UML component, deployment, and package diagrams generally appear in software architecture documents; UML class, object, and behavior diagrams appear in detailed functional design documents.

Examples of architectural styles and patterns

There are many common ways of designing computer software modules and their communications, among them:

- Blackboard

- Client–server model (2-tier, n-tier, peer-to-peer, cloud computing all use this model)
- Database-centric architecture (broad division can be made for programs which have database at its center and applications which don't have to rely on databases, E.g. desktop application programs, utility programs etc.)
- Distributed computing
- Event-driven architecture
- Front end and back end
- Implicit invocation
- Monolithic application
- Peer-to-peer
- Pipes and filters
- Plug-in (computing)
- Representational State Transfer
- Rule evaluation
- Search-oriented architecture (A pure SOA implements a service for every data access point.)
- Service-oriented architecture
- Shared nothing architecture
- Software componentry
- Space based architecture
- Structured (module-based but usually monolithic within modules)
- Three-tier model (An architecture with Presentation, Business Logic and Database tiers)

Chapter-4

Distributed Lock Manager and Distributed Database

Distributed lock manager

A **distributed lock manager** (**DLM**) provides distributed software applications with a means to synchronize their accesses to shared resources.

DLMs have been used as the foundation for several successful clustered file systems, in which the machines in a cluster can use each other's storage via a unified file system, with significant advantages for performance and availability. The main performance benefit comes from solving the problem of disk cache coherency between participating computers. The DLM is used not only for file locking but also for coordination of all disk access. VMScluster, the first clustering system to come into widespread use, relies on the OpenVMS DLM in just this way.

VMS implementation

VMS was the first widely-available operating system to implement a DLM. This became available in Version 4, although the user interface was the same as the single-processor lock manager that was first implemented in Version 3.

Resources

The DLM uses a generalised concept of a **resource**, which is some entity to which shared access must be controlled. This can relate to a file, a record, an area of shared memory, or anything else that the application designer chooses. A hierarchy of resources may be defined, so that a number of levels of locking can be implemented. For instance, a hypothetical database might define a resource hierarchy as follows:

- Database
- Table
- Record
- Field

A process can then acquire locks on the database as a whole, and then on particular parts of the database. A lock must be obtained on a parent resource before a subordinate resource can be locked.

Lock modes

A process running within a VMSCluster may obtain a lock on a resource. There are six lock modes that can be granted, and these determine the level of exclusivity of access to the resource. Once a lock has been granted, it is possible to convert the lock to a higher or lower level of lock mode. When all processes have unlocked a resource, the system's information about the resource is destroyed.

- Null Lock (NL). Indicates interest in the resource, but does not prevent other processes from locking it. It has the advantage that the resource and its lock value block are preserved, even when no processes are locking it.

- Concurrent Read (CR). Indicates a desire to read (but not update) the resource. It allows other processes to read or update the resource, but prevents others from having exclusive access to it. This is usually employed on high-level resources, in order that more restrictive locks can be obtained on subordinate resources.

- Concurrent Write (CW). Indicates a desire to read and update the resource. It also allows other processes to read or update the resource, but prevents others from having exclusive access to it. This is also usually employed on high-level resources, in order that more restrictive locks can be obtained on subordinate resources.

- Protected Read (PR). This is the traditional *share lock*, which indicates a desire to read the resource but prevents other from updating it. Others can however also read the resource.

- Protected Write (PW). This is the traditional *update lock*, which indicates a desire to read and update the resource and prevents others from updating it. Others with Concurrent Read access can however read the resource.

- Exclusive (EX). This is the traditional *exclusive lock* which allows read and update access to the resource, and prevents others from having any access to it.

The following truth table shows the compatibility of each lock mode with the others:

Mode	NL	CR	CW	PR	PW	EX
NL	Yes	Yes	Yes	Yes	Yes	Yes
CR	Yes	Yes	Yes	Yes	Yes	No
CW	Yes	Yes	Yes	No	No	No
PR	Yes	Yes	No	Yes	No	No

PW	Yes	Yes	No	No	No	No
EX	Yes	No	No	No	No	No

Obtaining a lock

A process can obtain a lock on a resource by *enqueueing* a lock request. This is similar to the QIO technique that is used to perform I/O. The enqueue lock request can either complete synchronously, in which case the process waits until the lock is granted, or asynchronously, in which case an AST occurs when the lock has been obtained.

It is also possible to establish a *blocking AST*, which is triggered when a process has obtained a lock that is preventing access to the resource by another process. The original process can then optionally take action to allow the other access (e.g. by demoting or releasing the lock).

Lock value block

A lock value block is associated with each resource. This can be read by any process that has obtained a lock on the resource (other than a null lock) and can be updated by a process that has obtained a protected update or exclusive lock on it.

It can be used to hold any information about the resource that the application designer chooses. A typical use is to hold a *version number* of the resource. Each time the associated entity (e.g. a database record) is updated, the holder of the lock increments the lock value block. When another process wishes to read the resource, it obtains the appropriate lock and compares the current lock value with the value it had last time the process locked the resource. If the value is the same, the process knows that the associated entity has not been updated since last time it read it, and therefore it is unnecessary to read it again. Hence, this technique can be used to implement various types of cache in a database or similar application.

Deadlock detection

When one or more processes have obtained locks on resources, it is possible to produce a situation where each is preventing another from obtaining a lock, and none of them can proceed. This is known as a *deadly embrace* or deadlock.

A simple example is when Process 1 has obtained an exclusive lock on Resource A, and Process 2 has obtained an exclusive lock on Resource B. If Process 1 then tries to lock Resource B, it will have to wait for Process 2 to release it. But if Process 2 then tries to lock Resource A, both processes will wait forever for each other.

The OpenVMS DLM periodically checks for deadlock situations. In the example above, the second lock enqueue request of one of the processes would return with a deadlock status. It would then be up to this process to take action to resolve the deadlock — in this case by releasing the first lock it obtained.

Linux clustering

Both Red Hat and Oracle have developed clustering software for Linux.

OCFS2, the Oracle Cluster File System was added to the official Linux kernel with version 2.6.16, in January 2006. The alpha-quality code warning on OCFS2 was removed in 2.6.19.

Red Hat's cluster software, including their DLM and Global File System was officially added to the Linux kernel with version 2.6.19, in November 2006.

Both systems use a DLM modeled on the venerable VMS DLM. Oracle's DLM has a simpler API. (the core function, `dlmlock()`, has eight parameters, whereas the VMS `SYS$ENQ` service and Red Hat's `dlm_lock` both have 11.)

Google's Chubby lock service

Google has developed *Chubby*, a lock service for loosely-coupled distributed systems. It is designed for coarse-grained locking and also provides a limited but reliable distributed file system. Key parts of Google's infrastructure, including Google File System, BigTable, and MapReduce, use Chubby to synchronize accesses to shared resources. Though Chubby was designed as a lock service, it is now heavily used inside Google as a name server, supplanting DNS.

SSI Systems

A DLM is also a key component of more ambitious single system image projects such as OpenSSI.

Distributed database

A **distributed database** is a database that is under the control of a central database management system (DBMS) in which storage devices are not all attached to a common CPU. It may be stored in multiple computers located in the same physical location, or may be dispersed over a network of interconnected computers.

Collections of data (e.g. in a database) can be distributed across multiple physical locations. A distributed database can reside on network servers on the Internet, on corporate intranets or extranets, or on other company networks. The replication and distribution of databases improves database performance at end-user worksites.

To ensure that the distributive databases are up to date and current, there are two processes: replication and duplication. Replication involves using specialized software that looks for changes in the distributive database. Once the changes have been identified, the replication process makes all the databases look the same. The replication process can be very complex and time consuming depending on the size and number of the distributive databases. This process can also require a lot of time and computer resources. Duplication on the other hand is not as complicated. It basically identifies one database as a master and then duplicates that database. The duplication process is normally done at a set time after hours. This is to ensure that each distributed location has the same data. In the duplication process, changes to the master database only are allowed. This is to ensure that local data will not be overwritten. Both of the processes can keep the data current in all distributive locations.

Besides distributed database replication and fragmentation, there are many other distributed database design technologies. For example, local autonomy, synchronous and asynchronous distributed database technologies. These technologies' implementation can and does depend on the needs of the business and the sensitivity/confidentiality of the data to be stored in the database, and hence the price the business is willing to spend on ensuring data security, consistency and integrity.

Basic architecture

A database User accesses the distributed database through:

Local applications
: applications which do not require data from other sites.

Global applications
: applications which do require data from other sites.

A distributed database does not share main memory or disks.

Important considerations

Care with a distributed database must be taken to ensure the following:

- The distribution is transparent — users must be able to interact with the system as if it were one logical system. This applies to the system's performance, and methods of access among other things.
- Transactions are transparent — each transaction must maintain database integrity across multiple databases. Transactions must also be divided into subtransactions, each subtransaction affecting one database system.

Advantages of distributed databases

- Management of distributed data with different levels of transparency.
- Increase reliability and availability.
- Easier expansion.

- Reflects organizational structure — database fragments are located in the departments they relate to.
- Local autonomy — a department can control the data about them (as they are the ones familiar with it.)
- Protection of valuable data — if there were ever a catastrophic event such as a fire, all of the data would not be in one place, but distributed in multiple locations.
- Improved performance — data is located near the site of greatest demand, and the database systems themselves are parallelized, allowing load on the databases to be balanced among servers. (A high load on one module of the database won't affect other modules of the database in a distributed database.)
- Economics — it costs less to create a network of smaller computers with the power of a single large computer.
- Modularity — systems can be modified, added and removed from the distributed database without affecting other modules (systems).
- Reliable transactions - Due to replication of database.
- Hardware, Operating System, Network, Fragmentation, DBMS, Replication and Location Independence.
- Continuous operation...

- Distributed Query processing.
- Distributed Transaction management.

Single site failure does not affect performance of system. All transactions follow A.C.I.D. property: a-atomicity, the transaction takes place as whole or not at all; c-consistency, maps one consistent DB state to another; i-isolation, each transaction sees a consistent DB; d-durability, the results of a transaction must survive system failures. The Merge Replication Method used to consolidate the data between databases.

Disadvantages of distributed databases

- Complexity — extra work must be done by the DBAs to ensure that the distributed nature of the system is transparent. Extra work must also be done to maintain multiple disparate systems, instead of one big one. Extra database design work must also be done to account for the disconnected nature of the database — for example, joins become prohibitively expensive when performed across multiple systems.
- Economics — increased complexity and a more extensive infrastructure means extra labour costs.
- Security — remote database fragments must be secured, and they are not centralized so the remote sites must be secured as well. The infrastructure must also be secured (e.g., by encrypting the network links between remote sites).
- Difficult to maintain integrity — but in a distributed database, enforcing integrity over a network may require too much of the network's resources to be feasible.,
- Inexperience — distributed databases are difficult to work with, and as a young field there is not much readily available experience on proper practice.

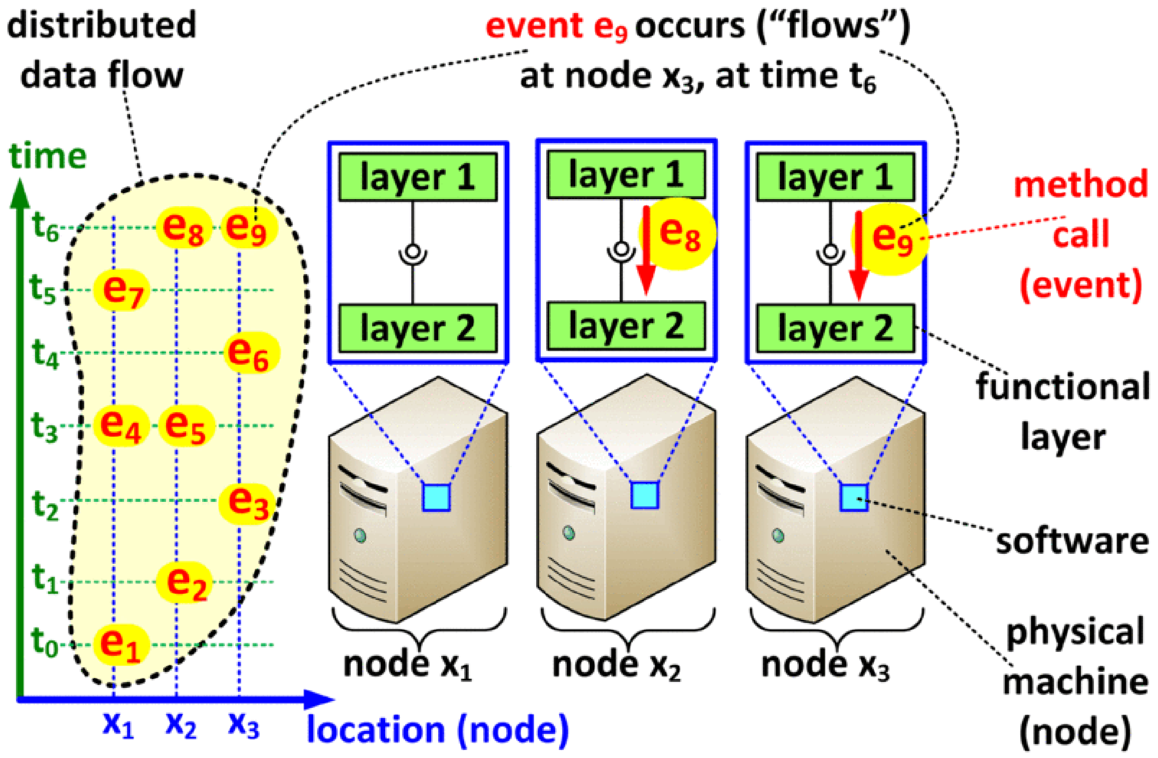
- Lack of standards — there are no tools or methodologies yet to help users convert a centralized DBMS into a distributed DBMS.
- Database design more complex — besides of the normal difficulties, the design of a distributed database has to consider fragmentation of data, allocation of fragments to specific sites and data replication.
- Additional software is required.
- Operating System should support distributed environment.
- Concurrency control: it is a major issue. It is solved by locking and timestamping.

Chapter-5

Distributed Data Flow and Client–server Model

Distributed data flow

Definitions



An illustration of the basic concepts involved in the definition of a distributed data flow.

The term *distributed data flow* (also abbreviated as *distributed flow*) refers to a set of events in a distributed application or protocol that satisfies the following informal properties:

- **Asynchronous**, **non-blocking**, and **one-way**. Each event represents a single instance of a non-blocking, one-way, asynchronous method invocation or other form of explicit or implicit message passing between two layers or software components. For example, each event might represent a single request to multicast a packet, issued by an application layer to an underlying multicast protocol. The requirement that events are one-way and asynchronous is important. Invocations of methods that may return results would normally be represented as two separate flows: one flow that represents the requests, and another flow that represents responses.

- **Homogeneous**, **unidirectional**, and **uniform**. All events in the distributed flow serve the same functional and logical purpose, and are related to one-another; generally, we require that they represent method calls or message exchanges between instances of the same functional layers, or instances of the same components, but perhaps on different nodes within a computer network. Furthermore, all events must flow in the same direction (i.e., one type of a layer or component always produces, and the other always consumes the events), and carry the same type of a payload. For example, a set of events that includes all multicast requests issued by the same application layer to the same multicast protocol is a distributed flow. On the other hand, a set of events that includes multicast requests made by different applications to different multicast protocols would not be considered a distributed flow, and neither would be a set of events that represent multicast requests as well as acknowledgments and error notifications.

- **Concurrent**, **continuous**, and **distributed**. The flow usually includes all events that flow between the two layers of software, simultaneously at different locations, and over a finite or infinite period of time. Thus, in general, events in a distributed flow are distributed both in space (they occur at different nodes) and in time (they occur at different times). For example, the flow of multicast requests would include all such requests made by instances of the given application on different nodes; normally, such flow would include events that occur on all nodes participating in the given multicast protocol. A flow, in which all events occur at the same node would be considered degenerate.

Formally, we represent each event in a distributed flow as a quadruple of the form (x,t,k,v), where *x* is the location (e.g., the network address of a physical node) at which the event occurs, *t* is the time at which this happens, *k* is a version, or a sequence number identifying the particular event, and *v* is a value that represents the event payload (e.g., all the arguments passed in a method call). Each distributed flow is a (possibly infinite) set of such quadruples that satisfies the following three formal properties.

- For any finite point in time *t*, there can be only finitely many events in the flow that occur at time *t* or earlier. This implies that in which flow, one can always point to the point in time at which the flow originated. The flow itself can be infinite; in such case, at any point in time, eventually a new event will appear in the flow.

- For any pair of events *e_1* and *e_2* that occur at the same location, if e_1 occurs at an earlier time than e_2, then the version number in e_1 must also be smaller than that of e_2.

- For any pair of events *e_1* and *e_2* that occur at the same location, if the two events have the same version numbers, they must also have the same values.

In addition to the above, flows can have a number of additional properties.

- **Consistency**. A distributed flow is said to be *consistent* if events with the same version always have the same value, even if they occur at different locations. Consistent flows typically represent various sorts of global decisions made by the protocol or application.

- **Monotonicity**. A distributed flow is said to be *weakly monotonic* if for any pair of events e_1 and e_2 that occur at the same location, if e_1 has a smaller version than e_2, then e_1 must carry a smaller value than e_2. A distributed flow is said to be *strongly monotonic* (or simply *monotonic*) if this is true even for pairs of events e_1 and e_2 that occur at different locations. Strongly monotonic flows are always consistent. They typically represent various sorts of irreversible decisions. Weakly monotonic flows may or may not be consistent.

Distributed data flows serve a purpose analogous to variables or method parameters in programming languages such as Java, in that they can represent state that is stored or communicated by a layer of software. Unlike variables or parameters, which represent a unit of state that resides in a single location, distributed flows are dynamic and distributed: they simultaneously appear in multiple locations within the network at the same time. As such, distributed flows are a more natural way of modeling the semantics and inner workings of certain classes of distributed systems. In particular, the distributed data flow abstraction has been used as a convenient way of expressing the high-level logical relationships between parts of distributed protocols.

Client–server model

The **client–server model** of computing is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server machine is a host that is running one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests.

Description

The *client–server* characteristic describes the relationship of cooperating programs in an application. The server component provides a function or service to one or many clients, which initiate requests for such services.

Functions such as email exchange, web access and database access, are built on the client–server model. Users accessing banking services from their computer use a web browser client to send a request to a web server at a bank. That program may in turn forward the request to its own database client program that sends a request to a database server at another bank computer to retrieve the account information. The balance is returned to the bank database client, which in turn serves it back to the web browser client displaying the results to the user. The client–server model has become one of the central ideas of network computing. Many business applications being written today use the client–server model. So do the Internet's main application protocols, such as HTTP, SMTP, Telnet, and DNS.

The interaction between client and server is often described using sequence diagrams. Sequence diagrams are standardized in the Unified Modeling Language.

Specific types of clients include web browsers, email clients, and online chat clients.

Specific types of servers include web servers, ftp servers, application servers, database servers, name servers, mail servers, file servers, print servers, and terminal servers. Most web services are also types of servers.

Comparison to peer-to-peer architecture

Peer-to-peer networks involve two or more computers pooling individual resources such as disk drives, CD-ROMs and printers. These shared resources are available to every computer in the network. Each computer acts as both the client and the server which means all the computers on the network are equals, that is where the term peer-to-peer comes from.While a client-server network involves multiple clients connecting to a single, central server.The file server on a client-server network is a high capacity, high speed computer with a large hard disk capacity.

In the peer to peer network, a software applications can be installed on the single computer and shared by every computer in the network. They are also cheaper to set up because most desktop operating systems have the software required for the network installed by default.On the other hand client-server model works with any size or physical layout of LAN and doesn't tend to slow down with a heavy use.

Peer-to-peer networks are typically less secure than a client-server networks because security is handled by the individual computers, not on the network as a whole. The resources of the computers in the network can become overburdened as they have to support not only the workstation user, but also the requests from network users. It is also difficult to provide systemwide services because the desktop operating system typically used in this type of network is incapable of hosting the service.Where the client-server networks have a higher initial setup cost. It is possible to set up a server on a desktop computer, but it is recommended that businesses invest in enterprise-class hardware and software. They also require a greater level of expertise to configure and manage the server hardware and software.

Advantages

- In most cases, a client–server architecture enables the roles and responsibilities of a computing system to be distributed among several independent computers that are known to each other only through a network. This creates an additional advantage to this architecture: greater ease of maintenance. For example, it is possible to replace, repair, upgrade, or even relocate a server while its clients remain both unaware and unaffected by that change.
- All data is stored on the servers, which generally have far greater security controls than most clients. Servers can better control access and resources, to guarantee that only those clients with the appropriate permissions may access and change data.
- Since data storage is centralized, updates to that data are far easier to administer in comparison to a P2P paradigm. In the latter, data updates may need to be distributed and applied to each peer in the network, which is time-consuming as there can be thousands or even millions of peers.
- Many mature client–server technologies are already available which were designed to ensure security, friendliness of the user interface, and ease of use.
- It functions with multiple different clients of different capabilities.

Disadvantages

- As the number of simultaneous client requests to a given server increases, the server can become overloaded. Contrast that to a P2P network, where its aggregated bandwidth actually increases as nodes are added, since the P2P network's overall bandwidth can be roughly computed as the sum of the bandwidths of every node in that network.
- The client–server paradigm lacks the robustness of a good P2P network. Under client–server, should a critical server fail, clients’ requests cannot be fulfilled. In P2P networks, resources are usually distributed among many nodes. Even if one or more nodes depart and abandon a downloading file, for example, the remaining nodes should still have the data needed to complete the download.

Chapter-6

Autonomic Computing and Message Passing

Autonomic Computing

Autonomic Computing refers to the self-managing characteristics of distributed computing resources, adapting to unpredictable changes whilst hiding intrinsic complexity to operators and users. Started by IBM in 2001, this initiative's ultimate aim is to develop computer systems capable of self-management, to overcome the rapidly growing complexity of computing systems management, and to reduce the barrier that complexity poses to further growth. An autonomic system makes decisions on its own, using high-level policies; it will constantly check and optimize its status and automatically adapt itself to changing conditions. As widely reported in literature, an autonomic computing framework might be seen composed by Autonomic Components (AC) interacting with each other . An AC can be modeled in terms of two main control loops (local and global) with sensors (for self-monitoring), effectors (for self-adjustment), knowledge and planner/adapter for exploiting policies based on self- and environment awareness.

Driven by such vision, a variety of architectural frameworks based on “self-regulating” autonomic components has been recently proposed. A very similar trend has recently characterized significant research work in the area of multi-agent systems. However, most of these approaches are typically conceived with centralized or cluster-based server architectures in mind and mostly address the need of reducing management costs rather than the need of enabling complex software systems or providing innovative services.

Autonomy-oriented computation is a paradigm proposed by Jiming Liu in 2001 that uses artificial systems imitating social animals' collective behaviours to solve hard computational problems. For example, ant colony optimization could be studied in this paradigm.

The problem of growing complexity

Self-management means different things in different fields: The number of computing devices in use is forecasted to grow at 38% per annum and the average complexity of each is increasing. Currently this volume and complexity is managed by highly skilled humans; but the demand for skilled IT personnel is already outstripping supply, with labour costs exceeding equipment costs by a ratio of up to 18:1. Computing systems have brought great benefits of speed and automation but there is now an overwhelming economic need to automate their maintenance.

In "The Vision of Autonomic Computing", Kephart and Chess warn that the dream of interconnectivity of computing systems and devices could become the “nightmare of pervasive computing” in which architects are unable to anticipate, design and maintain the complexity of interactions. They state the essence of autonomic computing is system self-management, freeing administrators from low-level task management while delivering better system behavior.

A general problem of modern distributed computing systems is that their complexity, and in particular the complexity of their management, is becoming a significant limiting factor in their further development. Large companies and institutions are employing large-scale computer networks for communication and computation. The distributed applications running on these computer networks are diverse and deal with many different tasks, ranging from internal control processes to presenting web content and to customer support.

Additionally, Mobile computing is pervading these networks at an increasing speed: employees need to communicate with their companies while they are not in their office. They do so by using laptops, PDAs, or mobile phones with diverse forms of wireless technologies to access their companies' data.

This creates an enormous complexity in the overall computer network which is hard to control manually by human operators. Manual control is time-consuming, expensive, and error-prone. The manual effort needed to control a growing networked computer system tends to increase very quickly.

80% of such problems in infrastructure happen at the client specific application and database layer. Most 'autonomic' service providers guarantee only up to the basic plumbing layer (power, hardware, operating system, network and basic database parameters).

Autonomic systems

A possible solution could be to enable modern, networked computing systems to manage themselves without direct human intervention. The *Autonomic Computing Initiative* (ACI) aims at providing the foundation for autonomic systems. It is inspired by the autonomic nervous system of the human body. This nervous system controls important bodily functions (e.g. respiration, heart rate, and blood pressure) without any conscious intervention.

In a self-managing Autonomic System, the human operator takes on a new role: he or she does not control the system directly. Instead, she defines general policies and rules that serve as an input for the self-management process. For this process, IBM has defined the following four functional areas:

- **Self-Configuration**: Automatic configuration of components;
- **Self-Healing**: Automatic discovery, and correction of faults;
- **Self-Optimization**: Automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements;
- **Self-Protection**: Proactive identification and protection from arbitrary attacks.

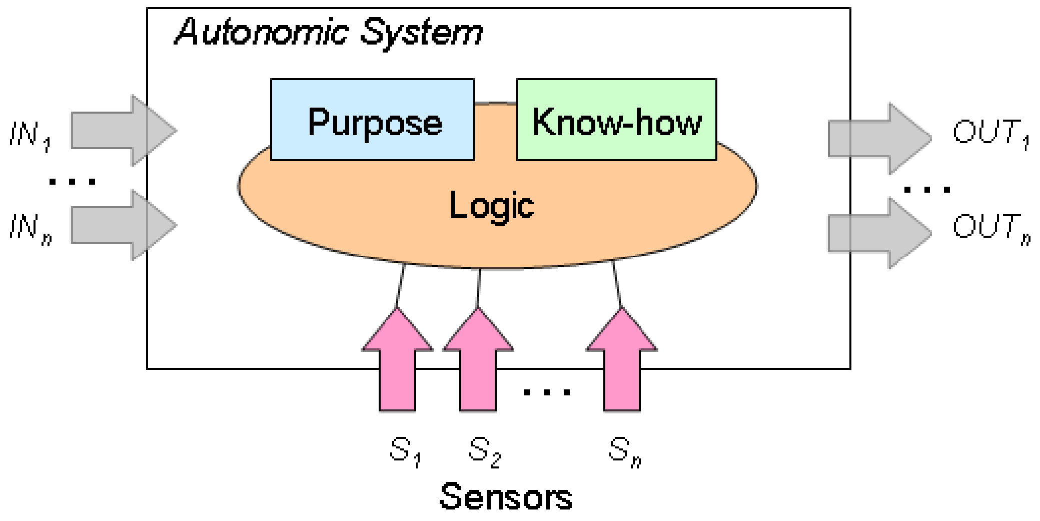
IBM defined five evolutionary levels, or the Autonomic deployment model, for its deployment: Level 1 is the basic level that presents the current situation where systems are essentially managed manually. Levels 2 - 4 introduce increasingly automated management functions, while level 5 represents the ultimate goal of autonomic, self-managing systems.

Control loops

A basic concept that will be applied in Autonomic Systems are closed control loops. This well-known concept stems from Process Control Theory. Essentially, a closed control loop in a self-managing system monitors some resource (software or hardware component) and autonomously tries to keep its parameters within a desired range.

According to IBM, hundreds or even thousands of these control loops are expected to work in a large-scale self-managing computer system.

Conceptual model



A fundamental building block of an autonomic system is the sensing capability (*Sensors S_i*), which enables the system to observe its external operational context. Inherent to an autonomic system is the knowledge of the *Purpose* (intension) and the *Know-how* to operate itself (e.g., bootstrapping, configuration knowledge, interpretation of sensory data, etc.) without external intervention. The actual operation of the autonomic system is dictated by the *Logic*, which is responsible for making the right decisions to serve its *Purpose*, and influence by the observation of the operational context (based on the sensor input).

This model highlights the fact that the operation of an autonomic system is purpose-driven. This includes its mission (e.g., the service it is supposed to offer), the policies (e.g., that define the basic behaviour), and the "survival instinct". If seen as a control system this would be encoded as a feedback error function or in a heuristically assisted system as an algorithm combined with set of heuristics bounding its operational space.

Characteristics

Even though the purpose and thus the behaviour of autonomic systems vary from system to system, every autonomic system should be able to exhibit a minimum set of properties to achieve its purpose:

Automatic
: This essentially means being able to self-control its internal functions and operations. As such, an autonomic system must be self-contained and able to start-up and operate without any manual intervention or external help. Again, the knowledge required to bootstrap the system (*Know-how*) must be inherent to the system.

Adaptive
: An autonomic system must be able to change its operation (i.e., its configuration, state and functions). This will allow the system to cope with temporal and spatial changes in its operational context either long term (environment customisation/optimisation) or short term (exceptional conditions such as malicious attacks, faults, etc.).

Aware
: An autonomic system must be able to monitor (sense) its operational context as well as its internal state in order to be able to assess if its current operation serves its purpose. Awareness will control adaptation of its operational behaviour in response to context or state changes.

Message passing

Message passing in computer science is a form of communication used in parallel computing, object-oriented programming, and interprocess communication. In this model, processes or objects can send and receive messages (comprising zero or more bytes, complex data structures, or even segments of code) to other processes. By waiting for messages, processes can also synchronize.

Overview

Message passing is the paradigm of communication where messages are sent from a sender to one or more recipients. Forms of messages include (remote) method invocation, signals, and data packets. When designing a message passing system several choices are made:

- Whether messages are transferred reliably
- Whether messages are guaranteed to be delivered in order
- Whether messages are passed one-to-one, one-to-many (multicasting or broadcasting), or many-to-one (client–server).
- Whether communication is synchronous or asynchronous.

Prominent theoretical foundations of concurrent computation, such as the Actor model and the process calculi are based on message passing. Implementations of concurrent systems that use message passing can either have message passing as an integral part of the language, or as a series of library calls from the language. Examples of the former include many distributed object systems. Examples of the latter include Microkernel operating systems pass messages between one kernel and one or more server blocks, and the Message Passing Interface used in high-performance computing.

Message passing systems and models

Distributed object and remote method invocation systems like ONC RPC, Corba, Java RMI, DCOM, SOAP, .NET Remoting, CTOS, QNX Neutrino RTOS, OpenBinder, D-Bus and similar are message passing systems.

Message passing systems have been called "shared nothing" systems because the message passing abstraction hides underlying state changes that may be used in the implementation of sending messages.

Message passing model based programming languages typically define messaging as the (usually asynchronous) sending (usually by copy) of a data item to a communication endpoint (Actor, process, thread, socket, *etc.*). Such messaging is used in Web Services by SOAP. This concept is the higher-level version of a datagram except that messages can be larger than a packet and can optionally be made reliable, durable, secure, and/or transacted.

Messages are also commonly used in the same sense as a means of interprocess communication; the other common technique being streams or pipes, in which data are sent as a sequence of elementary data items instead (the higher-level version of a virtual circuit).

Synchronous versus asynchronous message passing

Synchronous message passing systems require the sender and receiver to wait for each other to transfer the message. That is, the sender will not continue until the receiver has received the message.

Synchronous communication has two advantages. The first advantage is that reasoning about the program can be simplified in that there is a synchronisation point between sender and receiver on message transfer. The second advantage is that no buffering is required. The message can always be stored on the receiving side, because the sender will not continue until the receiver is ready.

Asynchronous message passing systems deliver a message from sender to receiver, without waiting for the receiver to be ready. The advantage of asynchronous communication is that the sender and receiver can overlap their computation because they do not wait for each other.

Synchronous communication can be built on top of asynchronous communication by ensuring that the sender always wait for an acknowledgement message from the receiver before continuing.

The buffer required in asynchronous communication can cause problems when it is full. A decision has to be made whether to block the sender or whether to discard future messages. If the sender is blocked, it may lead to an unexpected deadlock. If messages are dropped, then communication is no longer reliable.

Message passing versus calling

Message passing should be contrasted with the alternative communication method for passing information between programs - the Call. In a traditional `Call`, arguments are passed to the "callee" (the receiver) typically by one or more general purpose registers or in a parameter list containing the addresses of each of the arguments. This form of communication differs from message passing in at least three crucial areas -

- total memory usage
- transfer time
- locality

In message passing, each of the arguments has to have sufficient available *extra* memory for copying the existing argument into a portion of the new message. This applies irrespective of the size of the original arguments - so if one of the arguments is (say) an HTML string of 31,000 octets describing a web page, it has to be copied in its entirety (and perhaps even transmitted) to the receiving program (if not a local program).

By contrast, for the call method, only an address of say 4 or 8 bytes needs to be passed for each argument and may even be passed in a general purpose register requiring zero additional storage and zero "transfer time". This of course is not possible for distributed systems since an (absolute) address - in the callers address space - is normally meaningless to the remote program (however, a relative address might in fact be usable if the callee had an *exact* copy of, at least some of, the callers memory in advance). Web browsers and web servers are examples of processes that communicate by message passing. A URL is an example of a way of referencing resources that does depend on exposing the internals of a process.

A subroutine call or method invocation will not exit until the invoked computation has terminated. Asynchronous message passing, by contrast, can result in a response arriving a significant time after the request message was sent.

A message handler will, in general, process messages from more than one sender. This means its state can change for reasons unrelated to the behaviour of a single sender or client process. This is in contrast to the typical behaviour of an object upon which methods are being invoked: the latter is expected to remain in the same state between method invocations. (in other words, the message handler behaves analogously to a volatile object).

Message passing and locks

Message passing can be used as a way of controlling access to resources in a concurrent or asynchronous system. One of the main alternatives is mutual exclusion or locking. Examples of resources include shared memory, a disk file or region thereof, a database table or set of rows.

In locking, a resource is essentially shared, and processes wishing to access it (or a sector of it) must first obtain a lock. Once the lock is acquired, other processes are blocked out, ensuring that corruption from simultaneous writes does not occur. The lock is then released.

With the message-passing solution, it is assumed that the resource is not exposed, and all changes to it are made by an associated process, so that the resource is encapsulated. Processes wishing to access the resource send a request message to the handler. If the resource (or subsection) is available, the handler makes the requested change as an atomic event, that is conflicting requests are not acted on until the first request has been completed. If the resource is not available, the request is generally queued. The sending programme may or may not wait until the request has been completed.

Examples of message passing style

- Actor model implementation
- Amorphous computing
- Flow-based programming
- SOAP (protocol)

Influences on other programming models

In the terminology of some object-oriented programming languages, a message is the single means to pass control to an object. If the object "responds" to the message, it has a method for that message. In pure object-oriented programming, message passing is performed exclusively through a dynamic dispatch strategy.

Objects can send messages to other objects from within their method bodies. Message passing enables extreme late binding in systems. Sending the same message to an object twice will usually result in the object applying the method twice. Two messages are considered to be the same message type, if the name and the arguments of the message are identical. Some languages support the forwarding or delegation of method invocations from one object to another if the former has no method to handle the message, but "knows" another object that may have one.

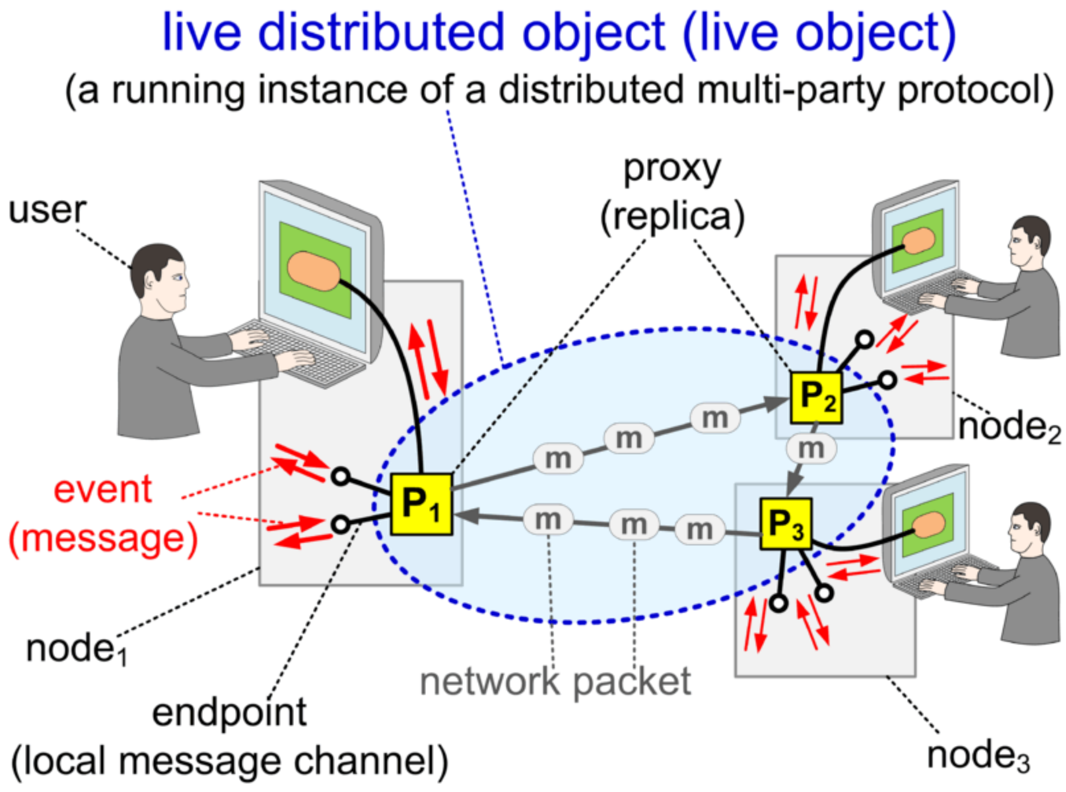
Alan Kay has argued that message passing is more important than objects in OOP, and that objects themselves are often over-emphasized. The live distributed objects programming model builds upon this observation; it uses the concept of a distributed data flow to characterize the behavior of a complex distributed system in terms of message patterns, using high-level, functional-style specifications.

Chapter-7

Live Distributed Object and High Level Architecture (simulation)

Live distributed object

Definitions



An illustration of the basic concepts involved in the definition of a live distributed object.

The term *live distributed object* (also abbreviated as *live object*) refers to a running instance of a distributed multi-party (or peer-to-peer) protocol, viewed from the object-oriented perspective, as an entity that has a distinct identity, may encapsulate internal state and threads of execution, and that exhibits a well-defined externally visible behavior. The key programming language concepts, as applied to live distributed objects, are defined as follows.

- **Identity**. The *identity* of a live distributed object is determined by the same factors that differentiate between instances of the same distributed protocol. The object consists of a group of software components physically executing on some set of physical machines and engaged in mutual communication, each executing the distributed protocol code with the same set of essential parameters, such as the name of a multicast group, the identifier of a publish-subscribe topic, the identity of a membership service, etc. Thus, for example, publish-subscribe channels and multicast groups are examples of live distributed objects: for each channel or group, there exists a single instance of a distributed protocol running among all computers sending, forwarding, or receiving the data published in the channel or multicast within the group. In this case, the object's identity is determined by the identifier of the channel or group, qualified with the identity of the distributed system that provides, controls, and manages the given channel or group. In the case of multicast, the identity of the system might be determined, for example, by the address of the *membership service* (the entity that manages the membership of the multicast group).

- **Proxies** (**replicas**). The *proxy* or a *replica* of a live object is one of the software component instances involved in executing the live object's distributed protocol. The object can thus be alternatively defined as a group of proxies engaged in communication, jointly maintaining some distributed state, and coordinating their operations. The term *proxy* stresses the fact that a single software component does not in itself constitute an object; rather, it serves as a *gateway* through which an application can gain access to a certain functionality or behavior that spans across a set of computers. In this sense, the concept of a live distributed object *proxy* generalizes the notion of a RPC, RMI, or .NET remoting client-side proxy stub.

- **Behavior**. The *behavior* of a live distributed object is characterized by the set of possible patterns of external interactions that its proxies can engage in with their local runtime environments. These interactions are modeled as exchanges of explicit events (messages).

- **State**. The *state* of a live distributed object is defined as the sum of all internal, local states of its proxies. By definition, it is distributed and replicated. The different replicas of the object's state may be strongly or only weakly consistent, depending on the protocol semantics: an instance of a consensus protocol will have the state of its replicas strongly consistent, whereas an instance of a leader election protocol will have a weakly consistent state. In this sense, the term *live distributed object* generalizes the concept of a *replicated object*; the latter is a specific type of live distributed object that uses a protocol such as Paxos, virtual synchrony, or state machine replication to achieve strong consistency between the internal states of its replicas. The state of a live distributed object should be understood as a dynamic notion: as a point (or *consistent cut*) in a stream of values, rather than as a particular value located in a given place at a given time. For example, the externally visible state of a leader election object would be defined as the identity of the currently elected leader. The identity is not stored at any particular location; rather, it materializes as a stream of messages of the form *elected(x)* concurrently produced by the proxies involved in executing this protocol, and concurrently consumed by instances of the application using this protocol, on different machines distributed across the network.

- **Interfaces** (**endpoints**). The *interface* of a live distributed object is defined by the types of interfaces exposed by its proxies; these may include event channels and various types of graphical user interfaces. Interfaces exposed by the proxies are referred to as the live distributed object's *endpoints*. The term *endpoint instance* refers to a single specific event channel or user interface exposed by a single specific proxy. To say that a live object *exposes* a certain endpoint means that each of its proxies exposes an instance of this endpoint to its local environment, and each of the endpoint instances carries events of the same types (or binds to the same type of a graphical display).

- **References**. The *reference* to a live object is a complete set of serialized, portable instructions for constructing its proxy. To *dereference* a reference means to locally parse and follow these instructions on a particular computer, to produce a running proxy of the live object. Defined this way, a live object reference plays the same role as a Java reference, a C/C++ pointer, or a web service's WSDL description; it contains a complete information sufficient to *locate* the given object and interact with it. Since live distributed objects may not reside in any particular place (but rather span across a dynamically changing set of computers), the information contained in a live distributed object's reference cannot be limited to just an address. If the object is identified by some sort of a gobally unique identifier (as might be the case for publish-subscribe topics or multicast groups), the reference must specify how this identifier is resolved, by recursively embedding a reference to the appropriate name resolution object.

- **Types**. The *type* of a live distributed object determines the patterns of external interactions with the object; it is determined by the types of endpoints and graphical user interfaces exposed by the object's proxies, and the patterns of events that may occur at the endpoints. The constraints that the object's type places on event patterns may span across the network. For example, type *atomic multicast* might specify that if an event of the form *deliver(x)* is generated by one proxy, a similar event must be eventually generated by all *non-faulty* proxies (proxies that run on computers that never crash, and that never cease to execute or are excluded from the protocol; the precise definition might vary). Much as it is the case for types in Java-like languages, there might exist many very different implementations of the same type. Thus, for example, behavior characteristic to *atomic multicast* might be exhibited by instances of distributed protocols such as virtual synchrony or Paxos.

The semantics and behavior of live distributed objects can be characterized in terms of distributed data flows; the set of messages or events that appear on the instances of a live object's endpoint forms a distributed data flow.

History

Early ideas underlying the concept of a live distributed object have been influenced by a rich body of research on object-oriented environments, programming language embeddings, and protocol composition frameworks, dating back at least to the actor model developed in the early 1970s; a comprehensive discussion of the relevant prior work can be found in Krzysztof Ostrowski's Ph.D. dissertation.

The term *live distributed object* was first used informally in a series of presentations given in the fall of 2006 at a ICWS conference , STC conference , and at the MSR labs in Redmond, WA , and then formally defined in 2007, in an IEEE Internet Computing article. Originally, the term was used to refer to the types of dynamic, interactive Web content that is not hosted on servers in data centers, but rather stored on the end-user's client computers, and internally powered by instances of reliable multicast protocols. The word *live* expressed the fact that the displayed information is dynamic, interactive, and represents current, fresh, live content that reflects recent updates made by the users (as opposed to static, read-only, and archival content that has been pre-assembled). The word *distributed* expressed the fact that the information is not hosted, stored at a server in a data center, but rather, it is replicated among the end-user computers, and updated in a peer-to-peer fashion through a stream of multicast messages that may be produced directly by the end-users consuming the content; a more comprehensive discussion of the live object concept in the context of Web development can be found in Krzysztof Ostrowski's Ph.D. dissertation.

The more general definition presented above has been first proposed in 2008, in a paper published at the ECOOP conference. The extension of the term has been motivated by the need to model live objects as compositions of other objects; in this sense, the concept has been inspired by Smalltalk, which pioneered the uniform perspective that *everything is an object*, and Jini, which pioneered the idea that *services are objects*. When applied to live distributed objects, the perspective dictates that their constituent parts, which includes instances of distributed multi-party protocols used internally to replicate state, should also be modeled as live distributed objects. The need for uniformity implies that the definition of a live distributed object must unify concepts such as live Web content, message streams, and instances of distributed multi-party protocols.

The first implementation of the live distributed object concept, as defined in the ECOOP paper, was the Live Distributed Objects platform developed by Krzysztof Ostrowski at Cornell University. The platform provided a set of visual, drag and drop tools for composing hierarchical documents resembling web pages, and containing XML-serialized live object references. Visual content such as chat windows, shared desktops, and various sorts of mashups could be composed by dragging and dropping components representing user interfaces and protocol instances onto a design form, and connecting them together. Since the moment of its creation, a number of extension have been developed to embed live distributed objects in Microsoft Office documents , and to support various types of hosted content such as Google Maps. As of March 2009, the platform is being actively developed by its creators.

High level architecture (simulation)

The **High Level Architecture** (**HLA**) is a general purpose architecture for distributed computer simulation systems. Using HLA, computer simulations can interact (that is, to communicate data, and to synchronize actions) to other computer simulations regardless of the computing platforms. The interaction between simulations is managed by a Run-Time Infrastructure (RTI).

Technical overview

A *High Level Architecture* consists of the following components:

- **Interface Specification**, that defines how HLA compliant simulators interact with the Run-Time Infrastructure (RTI). The RTI provides a programming library and an application programming interface (API) compliant to the interface specification.
- **Object Model Template** (OMT), that specifies what information is communicated between simulations, and how it is documented.
- **Rules**, that simulations must obey in order to be compliant to the standard..

Common HLA terminology

- **Federate**: an HLA compliant simulation entity.
- **Federation**: multiple simulation entities connected via the RTI using a common OMT.
- **Object**: a collection of related data sent between simulations.
- **Attribute**: data field of an object.
- **Interaction**: event sent between simulation entities.
- **Parameter**: data field of an interaction.

Interface specification

The interface specification is object oriented. Many RTIs provide APIs in C++ and the Java programming languages.

The interface specification is divided into service groups:

- Federation Management

- Declaration Management
- Object Management
- Ownership Management
- Time Management
- Data Distribution Management
- Support Services

Object model template

The object model template (OMT) provides a common framework for the communication between HLA simulations. OMT consists of the following documents:

- Federation Object Model (FOM). The FOM describes the shared object, attributes and interactions for the whole federation.
- Simulation Object Model (SOM). A SOM describes the shared object, attributes and interactions used for a single federate.

HLA rules

The HLA rules describe the responsibilities of federations and the federates that join.

1. Federations shall have an HLA Federation Object Model (FOM), documented in accordance with the HLA Object Model Template (OMT).
2. In a federation, all representation of objects in the FOM shall be in the federates, not in the run-time infrastructure (RTI).
3. During a federation execution, all exchange of FOM data among federates shall occur via the RTI.
4. During a federation execution, federates shall interact with the run-time infrastructure (RTI) in accordance with the HLA interface specification.
5. During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time.
6. Federates shall have an HLA Simulation Object Model (SOM), documented in accordance with the HLA Object Model Template (OMT).
7. Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.
8. Federates shall be able to transfer and/or accept ownership of an attribute dynamically during a federation execution, as specified in their SOM.
9. Federates shall be able to vary the conditions under which they provide updates of attributes of objects, as specified in their SOM.
10. Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation.

Base Object Model

The Base Object Model (BOM) is a new concept created by SISO to provide better reuse and composability for HLA simulations, and is highly relevant for HLA developers. More information can be found at Boms.info.

Federation Development and Execution Process (FEDEP)

FEDEP, IEEE 1516.3-2003, is a standardized and recommended process for developing interoperable HLA based federations. FEDEP is an overall framework overlay that can be used together with many other, commonly used development methodologies.

Distributed Simulation Engineering and Execution Process (DSEEP)

In spring 2007 SISO started revising the FEDEP. The upcoming version has been renamed to Distributed Simulation Engineering and Execution Process (DSEEP) and will also have a new standards number: IEEE 1730 instead of IEEE 1516.3. IEEE 1730 is expected to be published in 2010.

Standards

HLA is defined under IEEE Standard 1516:

- IEEE 1516-2010 - Standard for Modeling and Simulation High Level Architecture - Framework and Rules
- IEEE 1516.1-2010 - Standard for Modeling and Simulation High Level Architecture - Federate Interface Specification
- IEEE 1516.2-2010 - Standard for Modeling and Simulation High Level Architecture - Object Model Template (OMT) Specification
- IEEE 1516.3-2003 - Recommended Practice for High Level Architecture Federation Development and Execution Process (FEDEP)
- IEEE 1516.4-2007 - Recommended Practice for Verification, Validation, and Accreditation of a Federation an Overlay to the High Level Architecture Federation Development and Execution Process

Machine-readable parts of the standard, such as XML Schemas, C++, Java and WSDL APIs as well as FOM/SOM samples can be downloaded from the IEEE 1516 download area of the IEEE web site. The full standards texts are available at no extra cost to SISO members or can be purchased from the IEEE shop.

Previous version:

- IEEE 1516-2000 - Standard for Modeling and Simulation High Level Architecture - Framework and Rules
- IEEE 1516.1-2000 - Standard for Modeling and Simulation High Level Architecture - Federate Interface Specification

- IEEE 1516.1-2000 Errata (2003-oct-16)
- IEEE 1516.2-2000 - Standard for Modeling and Simulation High Level Architecture - Object Model Template (OMT) Specification

STANAG 4603

HLA (in both the current IEEE 1516 version and its ancestor "1.3" version) is the subject of the NATO standardization agreement (STANAG 4603) for modeling and simulation: *Modeling And Simulation Architecture Standards For Technical Interoperability: High Level Architecture (HLA)*.

DLC API

SISO has developed a complementary HLA API specification known as the Dynamic Link Compatible (DLC) API. The DLC API addresses a limitation of the IEEE 1516 and 1.3 API specification, whereby federate recompilation was necessary for each different RTI implementation. Note that this API has since been superseeded by the HLA Evolved APIs, informally known as Evolved DLC APIs (EDLC).

- SISO-STD-004-2004 - Dynamic Link Compatible HLA API Standard for the HLA Interface Specification Version 1.3
- SISO-STD-004.1-2004 - Dynamic Link Compatible HLA API Standard for the HLA Interface Specification (IEEE 1516.1 Version)

HLA Evolved

The IEEE 1516 standard has been revised under the SISO HLA-Evolved Product Development Group and was approved 25-Mar-2010 by the IEEE Standards Activities Board. The revised IEEE 1516-2010 standard includes current DoD standard interpretations and the EDLC API, an extended version of the SISO DLC API. Other major improvements include:

- Extended XML support for FOM/SOM, such as Schemas and extensibility
- Fault tolerance support services
- Web Services (WSDL) support/API
- Modular FOMs
- Update rate reduction
- Encoding helpers
- Extended support for additional transportation (such as QoS, IPv6,...)
- Standardized time representations

Chapter-8

Multitier Architecture, Utility Computing and Volunteer Computing

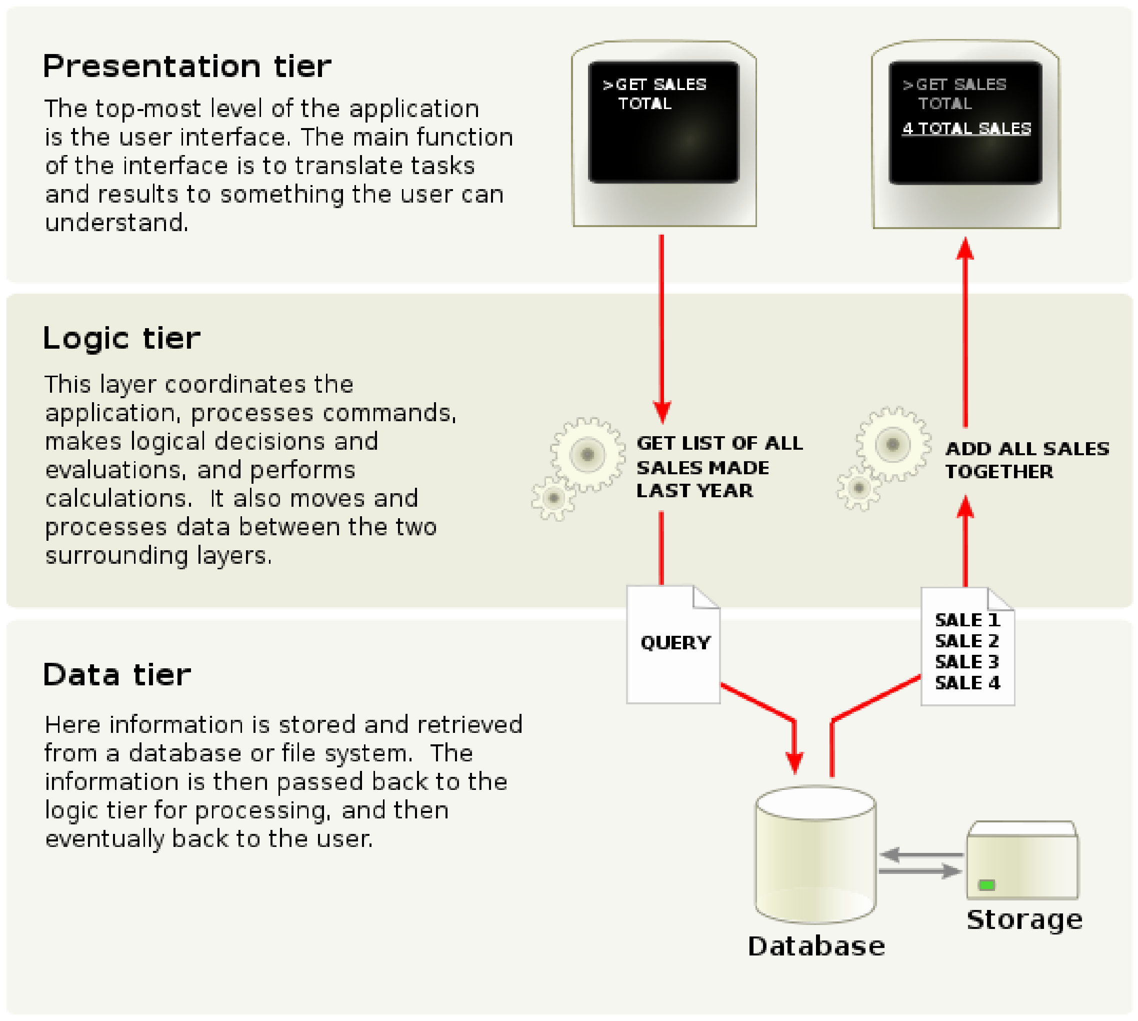
Multitier architecture

In software engineering, **multi-tier architecture** (often referred to as **n-tier architecture**) is a client–server architecture in which the presentation, the application processing, and the data management are logically separate processes. For example, an application that uses middleware to service data requests between a user and a database employs multi-tier architecture. The most widespread use of multi-tier architecture is the **three-tier architecture**.

N-tier application architecture provides a model for developers to create a flexible and reusable application. By breaking up an application into tiers, developers only have to modify or add a specific layer, rather than have to rewrite the entire application over. There should be a presentation tier, a business or data access tier, and a data tier.

The concepts of layer and tier are often used interchangeably. However, one fairly common point of view is that there is indeed a difference, and that a layer is a logical structuring mechanism for the elements that make up the software solution, while a tier is a physical structuring mechanism for the system infrastructure.

Three-tier architecture


Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
Database
Storage

Visual overview of a Three-tiered application

Three-tier is a client–server architecture in which the user interface, functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms. It was developed by John J. Donovan in Open Environment Corporation (OEC), a tools company he founded in Cambridge, Massachusetts.

The three-tier model is a software architecture and a software design pattern.

Apart from the usual advantages of modular software with well-defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently as requirements or technology change. For example, a change of operating system in the *presentation tier* would only affect the user interface code.

Typically, the user interface runs on a desktop PC or workstation and uses a standard graphical user interface, functional process logic may consist of one or more separate modules running on a workstation or application server, and an RDBMS on a database server or mainframe contains the computer data storage logic. The middle tier may be multi-tiered itself (in which case the overall architecture is called an "n-tier architecture").

Three-tier architecture has the following three tiers:

Presentation tier
: This is the topmost level of the application. The presentation tier displays information related to such services as browsing merchandise, purchasing, and shopping cart contents. It communicates with other tiers by outputting results to the browser/client tier and all other tiers in the network.

Application tier (business logic, logic tier, data access tier, or middle tier)
: The logic tier is pulled out from the presentation tier and, as its own layer, it controls an application's functionality by performing detailed processing.

Data tier
: This tier consists of database servers. Here information is stored and retrieved. This tier keeps data neutral and independent from application servers or business logic. Giving data its own tier also improves scalability and performance.

Comparison with the MVC architecture

At first glance, the three tiers may seem similar to the model-view-controller (MVC) concept; however, topologically they are different. A fundamental rule in a three tier architecture is the client tier never communicates directly with the data tier; in a three-tier model all communication must pass through the middle tier. Conceptually the three-tier architecture is linear. However, the MVC architecture is triangular: the view sends updates to the controller, the controller updates the model, and the view gets updated directly from the model.

From a historical perspective the three-tier architecture concept emerged in the 1990s from observations of distributed systems (e.g., web applications) where the client, middle ware and data tiers ran on physically separate platforms. Whereas MVC comes from the previous decade (by work at Xerox PARC in the late 1970s and early 1980s) and is based on observations of applications that ran on a single graphical workstation; MVC was applied to distributed applications later in its history.

Web development usage

In the web development field, three-tier is often used to refer to websites, commonly electronic commerce websites, which are built using three tiers:

1. A front-end web server serving static content, and potentially some cached dynamic content. In web based application, Front End is the content rendered by the browser. The content may be static or generated dynamically.
2. A middle dynamic content processing and generation level application server, for example Java EE, ASP.NET, PHP, ColdFusion platform.

3. A back-end database, comprising both data sets and the database management system or RDBMS software that manages and provides access to the data.

Other considerations

Data transfer between tiers is part of the architecture. Protocols involved may include one or more of SNMP, CORBA, Java RMI, .NET Remoting, Windows Communication Foundation, sockets, UDP, web services or other standard or proprietary protocols. Often middleware is used to connect the separate tiers. Separate tiers often (but not necessarily) run on separate physical servers, and each tier may itself run on a cluster.

Traceability

The end-to-end traceability of data flows through n-tier systems is a challenging task which becomes more important when systems increase in complexity. The Application Response Measurement defines concepts and APIs for measuring performance and correlating transactions between tiers.

Comments

Generally, the term tiers is used to describe physical distribution of components of a system on separate servers, computers, or networks (processing nodes). A three-tier architecture then will have three processing nodes. Layers refer to a logical grouping of components which may or may not be physically located on one processing node.

Utility computing

Utility Computing is the packaging of computing resources, such as computation, storage and services, as a metered service similar to a traditional public utility (such as electricity, water, natural gas, or telephone network). This model has the advantage of a low or no initial cost to acquire computer resources; instead, computational resources are essentially rented - turning what was previously a need to purchase products (hardware, software and network bandwidth) into a service.

This repackaging of computing services became the foundation of the shift to "On Demand" computing, Software as a Service and Cloud Computing models that further propagated the idea of computing, application and network as a service.

There was some initial skepticism about such a significant shift. However, the new model of computing caught and eventually became mainstream with the publication of Nick Carr's book "The Big Switch".

IBM, HP and Microsoft were early leaders in the new field of Utility Computing with their business units and researchers working on the architecture, payment and development challenges of the new computing model. Google, Amazon and others started to take the lead in 2008, as they established their own utility services for computing, storage and applications.

Utility Computing can support grid computing which has the characteristic of very large computations or a sudden peaks in demand which are supported via a large number of computers.

"Utility computing" has usually envisioned some form of virtualization so that the amount of storage or computing power available is considerably larger than that of a single time-sharing computer. Multiple servers are used on the "back end" to make this possible. These might be a dedicated computer cluster specifically built for the purpose of being rented out, or even an under-utilized supercomputer. The technique of running a single calculation on multiple computers is known as distributed computing.

The term "grid computing" is often used to describe a particular form of distributed computing, where the supporting nodes are geographically distributed or cross administrative domains. To provide utility computing services, a company can "bundle" the resources of members of the public for sale, who might be paid with a portion of the revenue from clients.

One model, common among volunteer computing applications, is for a central server to dispense tasks to participating nodes, on the behest of approved end-users (in the commercial case, the paying customers). Another model, sometimes called the Virtual Organization (VO), is more decentralized, with organizations buying and selling computing resources as needed or as they go idle.

The definition of "utility computing" is sometimes extended to specialized tasks, such as web services.

History

Utility computing is not a new concept, but rather has quite a long history. Among the earliest references is:

> “ If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry. ”
>
> —John McCarthy, speaking at the MIT Centennial in 1961

IBM and other mainframe providers conducted this kind of business in the following two decades, often referred to as time-sharing, offering computing power and database storage to banks and other large organizations from their world wide data centers. To facilitate this business model, mainframe operating systems evolved to include process control facilities, security, and user metering. The advent of mini computers changed this business model, by making computers affordable to almost all companies. As Intel and AMD increased the power of PC architecture servers with each new generation of processor, data centers became filled with thousands of servers.

In the late 90's utility computing re-surfaced. InsynQ (), Inc. launched [on-demand] applications and desktop hosting services in 1997 using HP equipment. In 1998, HP set up the Utility Computing Division in Mountain View, CA, assigning former Bell Labs computer scientists to begin work on a computing power plant, incorporating multiple utilities to form a software stack. Services such as "IP billing-on-tap" were marketed. HP introduced the Utility Data Center in 2001. Sun announced the Sun Cloud service to consumers in 2000. In December 2005, Alexa launched Alexa Web Search Platform, a Web search building tool for which the underlying power is utility computing. Alexa charges users for storage, utilization, etc. There is space in the market for specific industries and applications as well as other niche applications powered by utility computing. For example, PolyServe Inc. offers a clustered file system based on commodity server and storage hardware that creates highly available utility computing environments for mission-critical applications including Oracle and Microsoft SQL Server databases, as well as workload optimized solutions specifically tuned for bulk storage, high-performance computing, vertical industries such as financial services, seismic processing, and content serving. The Database Utility and File Serving Utility enable IT organizations to independently add servers or storage as needed, retask workloads to different hardware, and maintain the environment without disruption.

In spring 2006 3tera announced its AppLogic service and later that summer Amazon launched Amazon EC2 (Elastic Compute Cloud). These services allow the operation of general purpose computing applications. Both are based on Xen virtualization software and the most commonly used operating system on the virtual computers is Linux, though Windows and Solaris are supported. Common uses include web application, SaaS, image rendering and processing but also general-purpose business applications.

Utility computing merely means "Pay and Use", with regards to computing power.

Volunteer computing

Volunteer computing is a type of distributed computing in which computer owners donate their computing resources (such as processing power and storage) to one or more "projects".

History

The first volunteer computing project was the Great Internet Mersenne Prime Search, which was started in January 1996. It was followed in 1997 by distributed.net. In 1997 and 1998 several academic research projects developed Java-based systems for volunteer computing; examples include Bayanihan, Popcorn, Superweb, and Charlotte.. Another similar concept is Sideband computing which let a user to share his computing power while he is online.

The term "volunteer computing" was coined by Luis F. G. Sarmenta, the developer of Bayanihan. It is also appealing for global efforts on social responsibility, or Corporate Social Responsibility as reported in a Harvard Business Review or used in the Responsible IT forum.

In 1999 the SETI@home and Folding@home projects were launched. These projects received considerable media coverage, and each one attracted several hundred thousand volunteers.

Between 1998 and 2002, several companies were formed with business models involving volunteer computing. Examples include Popular Power, Porivo, Entropia, and United Devices.

In 2002, the Berkeley Open Infrastructure for Network Computing (BOINC) opensource project was founded, and became the software running the largest public computing grid (World Community Grid) in 2007.

Middleware for volunteer computing

The client software of the early volunteer computing projects consisted of a single program that combined the scientific computation and the distributed computing infrastructure. This monolithic architecture was inflexible; for example, it was difficult to deploy new application versions.

More recently, volunteer computing has moved to middleware systems that provide a distributed computing infrastructure independently of the scientific computation. Examples include:

- The Berkeley Open Infrastructure for Network Computing (BOINC). BOINC is the most widely-used middleware system, and is currently used by the World Community Grid. It is open source (LGPL) and is developed by an NSF-funded research project located at the UC Berkeley Space Sciences Laboratory. It offers client software for Windows, Mac OS X, Linux, and other Unix variants.
- XtremWeb is used primarily as a research tool. It is developed by a group based at the University of Paris - South.
- Xgrid is developed by Apple. Its client and server components run only on Mac OS X.

- Grid MP is a commercial middleware platform developed by United Devices and has been used in volunteer computing projects including grid.org, World Community Grid, Cell Computing, and Hikari Grid.

Most of these systems have the same basic structure: a client program runs on the volunteer's computer. It periodically contacts project-operated servers over the Internet, requesting jobs and reporting the results of completed jobs. This "pull" model is necessary because many volunteer computers are behind firewalls that don't allow incoming connections. The system keeps track of each user's "credit", a numerical measure of how much work that user's computers have done for the project.

Volunteer computing systems must deal with several problematic aspects of the volunteered computers: their heterogeneity, their churn (that is, the arrival and departure of hosts), their sporadic availability, and the need to not interfere with their performance during regular use.

In addition, volunteer computing systems must deal with several related problems related to correctness:

- Volunteers are unaccountable and essentially anonymous.
- Some volunteer computers (especially those that are overclocked) occasionally malfunction and return incorrect results.
- Some volunteers intentionally return incorrect results or claim excessive credit for results.

One common approach to these problems is "replicated computing", in which each job is performed on at least two computers. The results (and the corresponding credit) are accepted only if they agree sufficiently.

Costs for volunteer computing participants

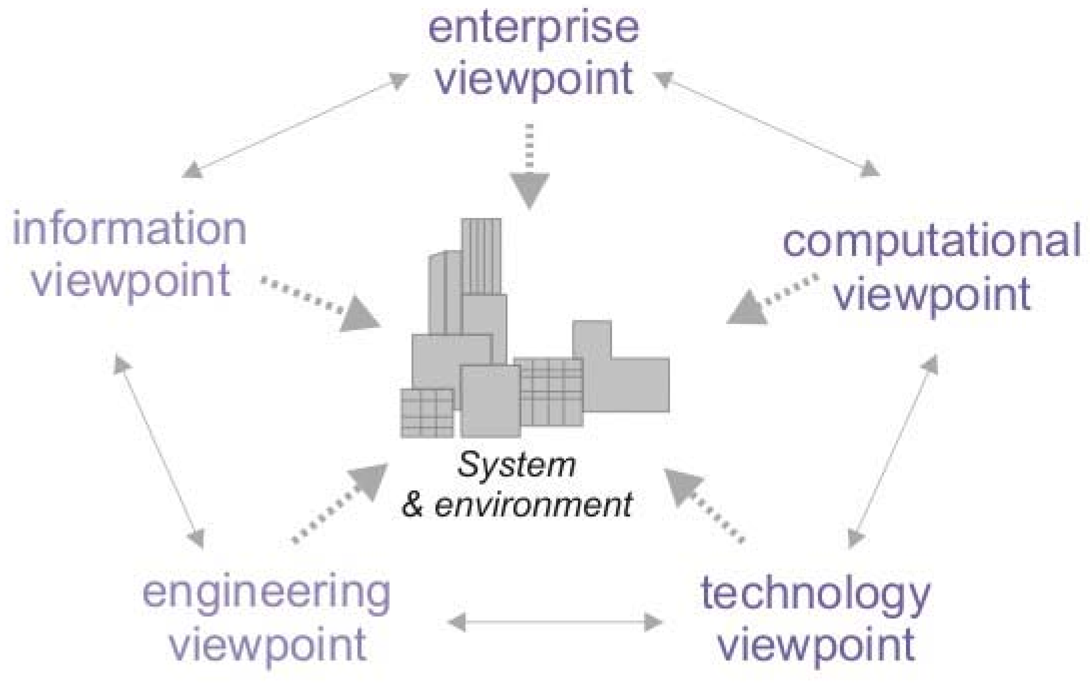
- Increased power consumption. A CPU that is idle generally has lower power consumption than when it is active. The desire to participate may also cause the volunteer to leave the PC on overnight, or to disable power-saving features like suspend. Additionally, if adequate cooling is not in place, this constant load on the volunteer's CPU can cause it to overheat.
- Decreased performance of the PC. If the volunteer computing application attempts to run while the computer is in use, it will impact performance of the PC. This is due to increased CPU contention, CPU cache contention, disk I/O contention, and network I/O contention. If RAM is a limitation, increased disk cache misses and/or increased paging can result. Volunteer computing applications typically execute at a lower CPU scheduling priority, which helps to alleviate CPU contention.

These effects may or may not be noticeable, and even if they are noticeable, the volunteer might choose to continue participating. However the increased power consumption can be remedied to some extent by setting the option of desired processor usage percent, that is available e.g. in BOINC client.

Chapter-9

RM-ODP

Reference Model of Open Distributed Processing (RM-ODP) is a reference model in computer science, which provides a co-ordinating framework for the standardization of open distributed processing (ODP). It supports distribution, interworking, platform and technology independence, and portability, together with an enterprise architecture framework for the specification of ODP systems.



The RM-ODP view model, which provides five generic and complementary viewpoints on the system and its environment.

RM-ODP, also named *ITU-T Rec. X.901-X.904* and *ISO/IEC 10746*, is a joint effort by the International Organization for Standardization (ISO), the International

Electrotechnical Commission (IEC) and the Telecommunication Standardization Sector (ITU-T).

Overview

The RM-ODP is a reference model based on precise concepts derived from current distributed processing developments and, as far as possible, on the use of formal description techniques for specification of the architecture. Many RM-ODP concepts, possibly under different names, have been around for a long time and have been rigorously described and explained in exact philosophy (for example, in the works of Mario Bunge) and in systems thinking (for example, in the works of Friedrich Hayek). Some of these concepts -- such as abstraction, composition, and emergence -- have recently been provided with a solid mathematical foundation in category theory.

RM-ODP has four fundamental elements:

- an object modelling approach to system specification;
- the specification of a system in terms of separate but interrelated viewpoint specifications;
- the definition of a system infrastructure providing distribution transparencies for system applications; and
- a framework for assessing system conformance.

The RM-ODP family of recommendations and international standards defines a system of interrelated essential concepts necessary to specify open distributed processing systems and provides a well-developed enterprise architecture framework for structuring the specifications for any large-scale systems including software systems.

History

Much of the preparatory work that led into the adoption of RM-ODP as an ISO standard was carried out by the Advanced Networked Systems Architecture (ANSA) project. This ran from 1984 until 1998 under the leadership of Andrew Herbert (now MD of Microsoft Research in Cambridge), and involved a number of major computing and telecommunication companies. Parts 2 and 3 of the RM-ODP were eventually adopted as ISO standards in 1996. Parts 1 and 4 were adopted in 1998.

RM-ODP Topics

RM-ODP standards

RM-ODP consists of four basic ITU-T Recommendations and ISO/IEC International Standards:

1. Overview : Contains a motivational overview of ODP, giving scoping, justification and explanation of key concepts, and an outline of the ODP

architecture. It contains explanatory material on how the RM-ODP is to be interpreted and applied by its users, who may include standard writers and architects of ODP systems.

2. Foundations : Contains the definition of the concepts and analytical framework for normalized description of (arbitrary) distributed processing systems. It introduces the principles of conformance to ODP standards and the way in which they are applied. In only 18 pages, this standard sets the basics of the whole model in a clear, precise and concise way.
3. Architecture : Contains the specification of the required characteristics that qualify distributed processing as open. These are the constraints to which ODP standards must conform. This recommendation also defines RM-ODP viewpoints, subdivisions of the specification of a whole system, established to bring together those particular pieces of information relevant to some particular area of concern.
4. Architectural Semantics : Contains a formalization of the ODP modeling concepts by interpreting many concepts in terms of the constructs of the different standardized formal description techniques.

Viewpoints modeling and the RM-ODP framework

Most complex system specifications are so extensive that no single individual can fully comprehend all aspects of the specifications. Furthermore, we all have different interests in a given system and different reasons for examining the system's specifications. A business executive will ask different questions of a system make-up than would a system implementer. The concept of RM-ODP viewpoints framework, therefore, is to provide separate viewpoints into the specification of a given complex system. These viewpoints each satisfy an audience with interest in a particular set of aspects of the system. Associated with each viewpoint is a viewpoint language that optimizes the vocabulary and presentation for the audience of that viewpoint.

Viewpoint modeling has become an effective approach for dealing with the inherent complexity of large distributed systems. Current software architectural practices, as described in IEEE 1471, divide the design activity into several areas of concerns, each one focusing on a specific aspect of the system. Examples include the "4+1" view model, the Zachman Framework, TOGAF, DoDAF and, of course, RM-ODP.

A viewpoint is a subdivision of the specification of a complete system, established to bring together those particular pieces of information relevant to some particular area of concern during the analysis or design of the system. Although separately specified, the viewpoints are not completely independent; key items in each are identified as related to items in the other viewpoints. Moreover, each viewpoint substantially uses the same foundational concepts (defined in Part 2 of RM-ODP). However, the viewpoints are sufficiently independent to simplify reasoning about the complete specification. The mutual consistency among the viewpoints is ensured by the architecture defined by RM-ODP, and the use of a common object model provides the glue that binds them all together.

More specifically, the RM-ODP framework provides five generic and complementary viewpoints on the system and its environment:

- The *enterprise viewpoint*, which focuses on the purpose, scope and policies for the system. It describes the business requirements and how to meet them.
- The *information viewpoint*, which focuses on the semantics of the information and the information processing performed. It describes the information managed by the system and the structure and content type of the supporting data.
- The *computational viewpoint*, which enables distribution through functional decomposition on the system into objects which interact at interfaces. It describes the functionality provided by the system and its functional decomposition.
- The *engineering viewpoint*, which focuses on the mechanisms and functions required to support distributed interactions between objects in the system. It describes the distribution of processing performed by the system to manage the information and provide the functionality.
- The *technology viewpoint*, which focuses on the choice of technology of the system. It describes the technologies chosen to provide the processing, functionality and presentation of information.

RM-ODP and UML

Currently there is growing interest in the use of UML for system modelling. However, there is no widely agreed approach to the structuring of such specifications. This adds to the cost of adopting the use of UML for system specification, hampers communication between system developers and makes it difficult to relate or merge system specifications where there is a need to integrate IT systems.

Although the ODP reference model provides abstract languages for the relevant concepts, it does not prescribe particular notations to be used in the individual viewpoints. The viewpoint languages defined in the reference model are abstract languages in the sense that they define what concepts should be used, not how they should be represented. This lack of precise notations for expressing the different models involved in a multi-viewpoint specification of a system is a common feature for most enterprise architectural approaches, including the Zachman Framework, the "4+1" model, or the RM-ODP. These approaches were consciously defined in a notation- and representation-neutral manner to increase their use and flexibility. However, this makes more difficult, among other things, the development of industrial tools for modeling the viewpoint specifications, the formal analysis of the specifications produced, and the possible derivation of implementations from the system specifications.

In order to address these issues, ISO/IEC and the ITU-T started a joint project in 2004: "ITU-T Rec. X.906|ISO/IEC 19793: Information technology - Open distributed processing - Use of UML for ODP system specifications". This document (usually referred to as UML4ODP) defines use of the Unified Modeling Language 2 (UML 2; ISO/IEC 19505), for expressing the specifications of open distributed systems in terms of the viewpoint specifications defined by the RM-ODP.

It defines a set of UML Profiles, one for each viewpoint language and one to express the correspondences between viewpoints, and an approach for structuring them according to the RM-ODP principles. The purpose of "UML4ODP" to allow ODP modelers to use the UML notation for expressing their ODP specifications in a standard graphical way; to allow UML modelers to use the RM-ODP concepts and mechanisms to structure their large UML system specifications according to a mature and standard proposal; and to allow UML tools to be used to process viewpoint specifications, thus facilitating the software design process and the enterprise architecture specification of large software systems.

In addition, ITU-T Rec. X.906 | ISO/IEC 19793 enables the seamless integration of the RM-ODP enterprise architecture framework with the Model-Driven Architecture (MDA initiative from the OMG, and with the service-oriented architecture (SOA).

Applications

In addition, there are several projects that have used or currently use RM-ODP for effectively structuring their systems specifications:

- The COMBINE project
- The Reference Architecture for Space Data Systems (RASDS) From the Consultative Committee for Space Data Systems.
- Interoperability Technology Association for Information Processing (INTAP), Japan.
- The Synapses European project.

Chapter-10

Supercomputer

The Columbia Supercomputer, located at the NASA Ames Research Center.

A 1985 supercomputer Cray-2

A **supercomputer** is a computer that is at the frontline of current processing capacity, particularly speed of calculation. Supercomputers were introduced in the 1960s and were designed primarily by Seymour Cray at Control Data Corporation (CDC), which led the market into the 1970s until Cray left to form his own company, Cray Research. He then took over the supercomputer market with his new designs, holding the top spot in supercomputing for five years (1985–1990). In the 1980s a large number of smaller competitors entered the market, in parallel to the creation of the minicomputer market a decade earlier, but many of these disappeared in the mid-1990s "supercomputer market crash".

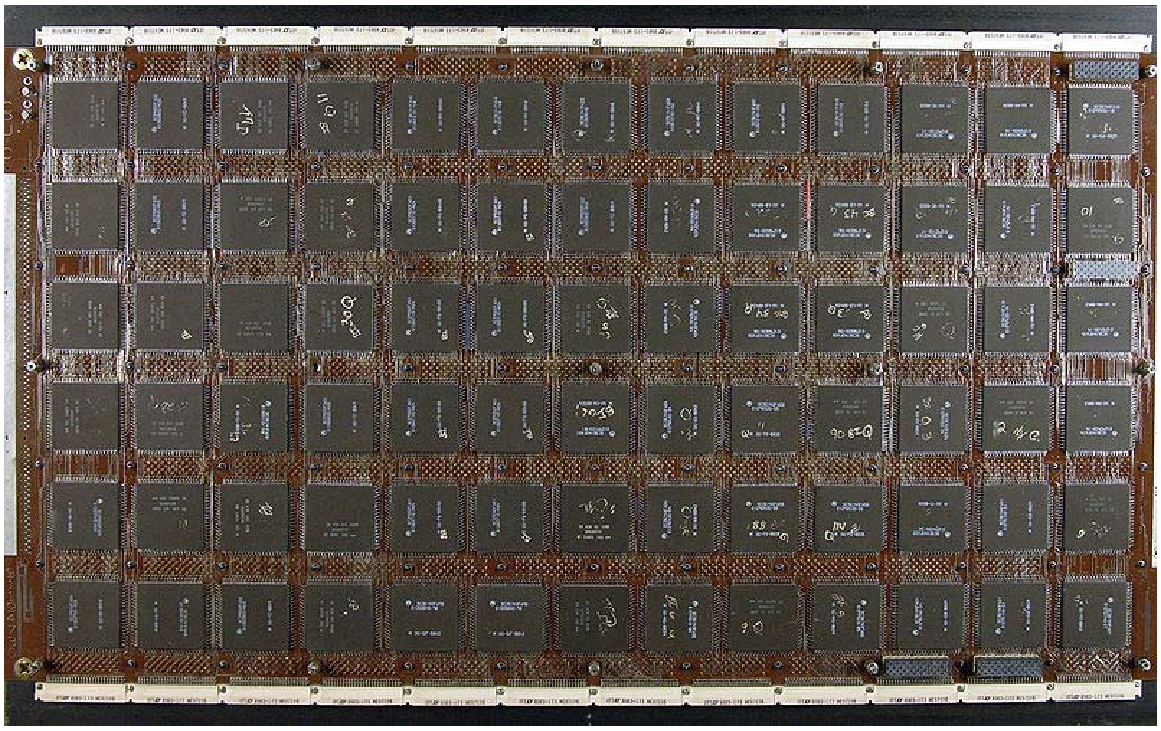
Today, supercomputers are typically one-of-a-kind custom designs produced by "traditional" companies such as Cray, IBM and Hewlett-Packard, who had purchased many of the 1980s companies to gain their experience. Since October 2010, the Tianhe-1A supercomputer has been the fastest in the world; it is located in China.

The term *supercomputer* itself is rather fluid, and the speed of today's supercomputers tends to become typical of tomorrow's ordinary computers. CDC's early machines were simply very fast scalar processors, some ten times the speed of the fastest machines offered by other companies. In the 1970s most supercomputers were dedicated to running a vector processor, and many of the newer players developed their own such processors at a lower price to enter the market. The early and mid-1980s saw machines with a modest number of vector processors working in parallel to become the standard. Typical numbers of processors were in the range of four to sixteen. In the later 1980s and 1990s, attention turned from vector processors to massive parallel processing systems with thousands of "ordinary" CPUs, some being off the shelf units and others being custom designs. Today, parallel designs are based on "off the shelf" server-class microprocessors, such as the PowerPC, Opteron, or Xeon, and coprocessors like NVIDIA Tesla GPGPUs, AMD GPUs, IBM Cell, FPGAs. Most modern supercomputers are now highly-tuned computer clusters using commodity processors combined with custom interconnects.

Supercomputers are used for highly calculation-intensive tasks such as problems involving quantum physics, weather forecasting, climate research, molecular modeling (computing the structures and properties of chemical compounds, biological macromolecules, polymers, and crystals), physical simulations (such as simulation of airplanes in wind tunnels, simulation of the detonation of nuclear weapons, and research into nuclear fusion).

Relevant here is the distinction between capability computing and capacity computing, as defined by Graham et al. **Capability computing** is typically thought of as using the maximum computing power to solve a large problem in the shortest amount of time. Often a capability system is able to solve a problem of a size or complexity that no other computer can. **Capacity computing** in contrast is typically thought of as using efficient cost-effective computing power to solve somewhat large problems or many small problems or to prepare for a run on a capability system.

Hardware and software design



Processor board of a CRAY YMP vector computer

Supercomputers using custom CPUs traditionally gained their speed over conventional computers through the use of innovative designs that allow them to perform many tasks in parallel, as well as complex detail engineering. They tend to be specialized for certain types of computation, usually numerical calculations, and perform poorly at more general computing tasks. Their memory hierarchy is very carefully designed to ensure the processor is kept fed with data and instructions at all times — in fact, much of the performance difference between slower computers and supercomputers is due to the memory hierarchy. Their I/O systems tend to be designed to support high bandwidth, with latency less of an issue, because supercomputers are not used for transaction processing.

As with all highly parallel systems, Amdahl's law applies, and supercomputer designs devote great effort to eliminating software serialization, and using hardware to address the remaining bottlenecks.

Supercomputer challenges, technologies

- A supercomputer consumes large amounts of electrical power, almost all of which is converted into heat, requiring cooling. For example, Tianhe-1A consumes 4.04 Megawatts of electricity. The cost to power and cool the system is usually one of the factors that limit the scalability of the system. (For example, 4MW at $0.10/KWh is $400 an hour or about $3.5 million per year).
- Information cannot move faster than the speed of light between two parts of a supercomputer. For this reason, a supercomputer that is many meters across must have latencies between its components measured at least in the tens of nanoseconds. Seymour Cray's supercomputer designs attempted to keep cable runs as short as possible for this reason, hence the cylindrical shape of his Cray range of computers. In modern supercomputers built of many conventional CPUs running in parallel, latencies of 1–5 microseconds to send a message between CPUs are typical.
- Supercomputers consume and produce massive amounts of data in a very short period of time. According to Ken Batcher, "A supercomputer is a device for turning compute-bound problems into I/O-bound problems." Much work on external storage bandwidth is needed to ensure that this information can be transferred quickly and stored/retrieved correctly.

Technologies developed for supercomputers include:

- Vector processing
- Liquid cooling
- Non-Uniform Memory Access (NUMA)
- Striped disks (the first instance of what was later called RAID)
- Parallel filesystems

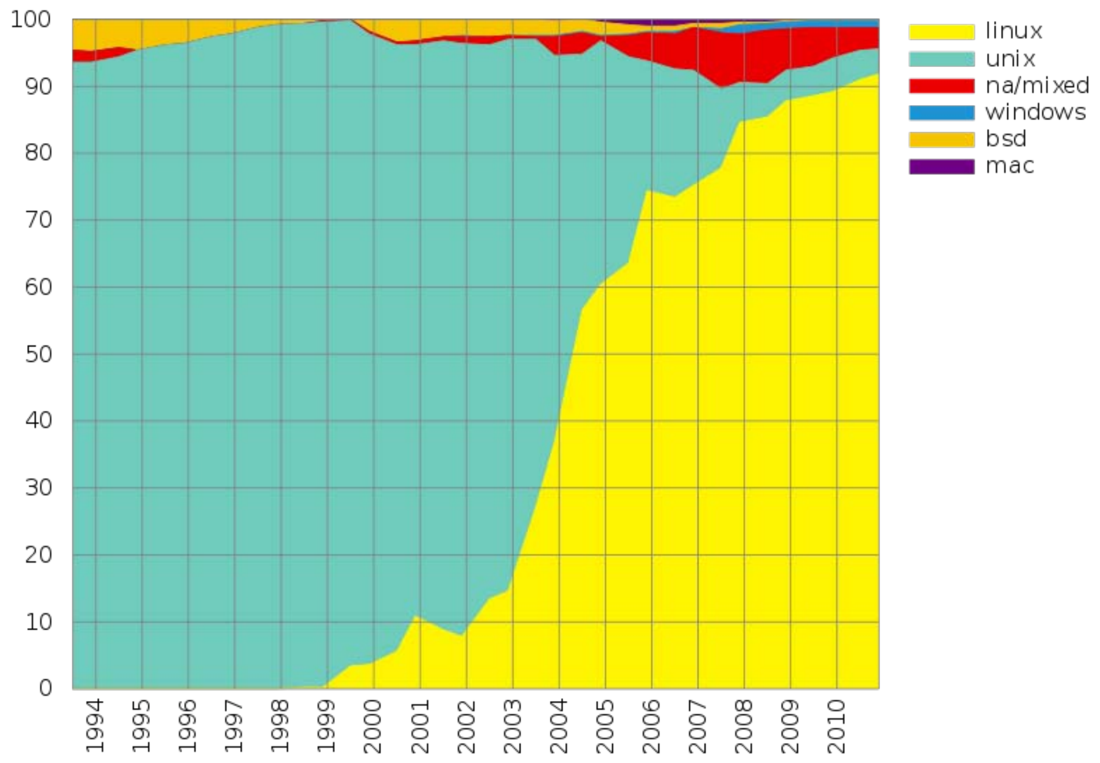
Processing techniques

Vector processing techniques were first developed for supercomputers and continue to be used in specialist high-performance applications. Vector processing techniques have trickled down to the mass market in DSP architectures and SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) processing instructions for general-purpose computers.

Modern video game consoles in particular use SIMD extensively and this is the basis for some manufacturers' claim that their game machines are themselves supercomputers. Indeed, some graphics cards have the computing power of several TeraFLOPS. The applications to which this power can be applied was limited by the special-purpose nature of early video processing. As video processing has become more sophisticated, graphics processing units (GPUs) have evolved to become more useful as general-purpose vector processors, and an entire computer science sub-discipline has arisen to exploit this capability: General-Purpose Computing on Graphics Processing Units (GPGPU).

The current Top500 list (from May 2010) has 3 supercomputers based on GPGPUs. In particular, the number 3 supercomputer, Nebulae built by Dawning in China, is based on GPGPUs.

Operating systems



More than 90% of today's Supercomputers run some variant of Linux.

Supercomputers today most often use variants of Linux. as shown by the graph to the right.

Until the early-to-mid-1980s, supercomputers usually sacrificed instruction set compatibility and code portability for performance (processing and memory access speed). For the most part, supercomputers to this time (unlike high-end mainframes) had vastly different operating systems. The Cray-1 alone had at least six different proprietary OSs largely unknown to the general computing community. In a similar manner, different and incompatible vectorizing and parallelizing compilers for Fortran existed. This trend would have continued with the ETA-10 were it not for the initial instruction set compatibility between the Cray-1 and the Cray X-MP, and the adoption of computer systems such as Cray's Unicos, or Linux.

Programming

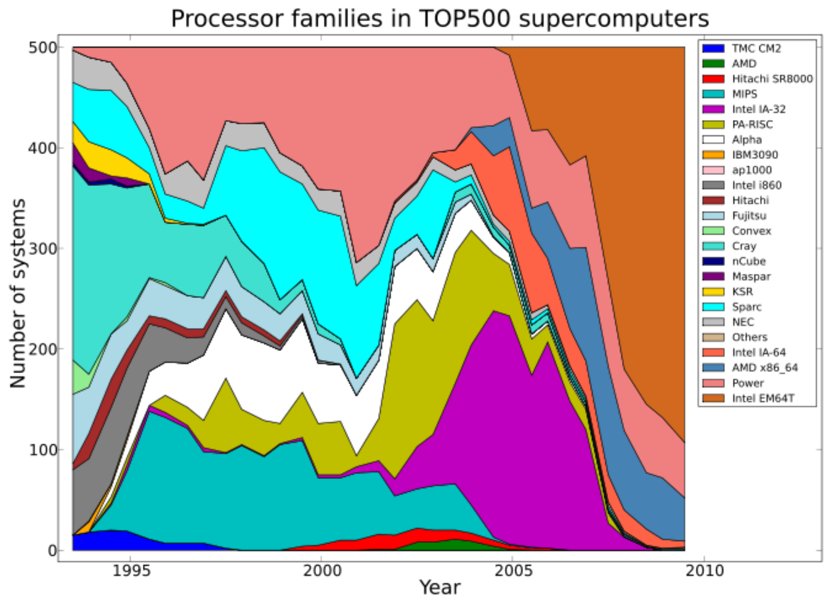
The parallel architectures of supercomputers often dictate the use of special programming techniques to exploit their speed. The base language of supercomputer code is, in general, Fortran or C, using special libraries to share data between nodes. In the most common scenario, environments such as PVM and MPI for loosely connected clusters and OpenMP for tightly coordinated shared memory machines are used. Significant effort is required to optimize a problem for the interconnect characteristics of the machine it will be run on; the aim is to prevent any of the CPUs from wasting time waiting on data from other nodes. The new massively parallel GPGPUs have hundreds of processor cores and are programmed using programming models such as CUDA and OpenCL.

Software tools

Software tools for distributed processing include standard APIs such as MPI and PVM, VTL, and open source-based software solutions such as Beowulf, WareWulf, and openMosix, which facilitate the creation of a supercomputer from a collection of ordinary workstations or servers. Technology like ZeroConf (Rendezvous/Bonjour) can be used to create ad hoc computer clusters for specialized software such as Apple's Shake compositing application. An easy programming language for supercomputers remains an open research topic in computer science. Several utilities that would once have cost several thousands of dollars are now completely free thanks to the open source community that often creates disruptive technology.

Modern supercomputer architecture

IBM Roadrunner – LANL



The CPU Architecture Share of Top500 Rankings between 1993 and 2009.

Supercomputers today often have a similar top-level architecture consisting of a cluster of MIMD multiprocessors, each processor of which is SIMD. The supercomputers vary radically with respect to the number of multiprocessors per cluster, the number of processors per multiprocessor, and the number of simultaneous instructions per SIMD processor. Within this hierarchy we have:

- A computer cluster is a collection of computers that are highly interconnected via a high-speed network or switching fabric. Each computer runs under a separate instance of an Operating System (OS).
- A multiprocessing computer is a computer, operating under a single OS and using more than one CPU, wherein the application-level software is indifferent to the number of processors. The processors share tasks using Symmetric multiprocessing (SMP) and Non-Uniform Memory Access (NUMA).
- A SIMD processor executes the same instruction on more than one set of data at the same time. The processor could be a general purpose commodity processor or special-purpose vector processor. It could also be high-performance processor or a low power processor. As of 2007, the processor executes several SIMD instructions per nanosecond.

As of October 2010 the fastest supercomputer in the world is the Tianhe-1A system at National University of Defense Technology with more than 21000 computers, it boasts a speed of 2.507 petaflops, over 30% faster than the world's next fastest computer, the Cray XT5 "Jaguar".

In February 2009, IBM also announced work on "Sequoia," which appears to be a 20 petaflops supercomputer. This will be equivalent to 2 million laptops (whereas Roadrunner is comparable to a mere 100,000 laptops). It is slated for deployment in late 2011. The Sequoia will be powered by 1.6 million cores (specific 45-nanometer chips in development) and 1.6 petabytes of memory. It will be housed in 96 refrigerators spanning roughly 3,000 square feet (280 m^2).

Moore's Law and economies of scale are the dominant factors in supercomputer design. The design concepts that allowed past supercomputers to out-perform desktop machines of the time tended to be gradually incorporated into commodity PCs. Furthermore, the costs of chip development and production make it uneconomical to design custom chips for a small run and favor mass-produced chips that have enough demand to recoup the cost of production. A current model quad-core Xeon workstation running at 2.66 GHz will outperform a multimillion dollar Cray C90 supercomputer used in the early 1990s; most workloads requiring such a supercomputer in the 1990s can be done on workstations costing less than 4,000 US dollars as of 2010. Supercomputing is taking a step of increasing density, allowing for desktop supercomputers to become available, offering the computer power that in 1998 required a large room to require less than a desktop footprint.

In addition, many problems carried out by supercomputers are particularly suitable for parallelization (in essence, splitting up into smaller parts to be worked on simultaneously) and, in particular, fairly coarse-grained parallelization that limits the amount of information that needs to be transferred between independent processing units. For this reason, traditional supercomputers can be replaced, for many applications, by "clusters" of computers of standard design, which can be programmed to act as one large computer.

Special-purpose supercomputers

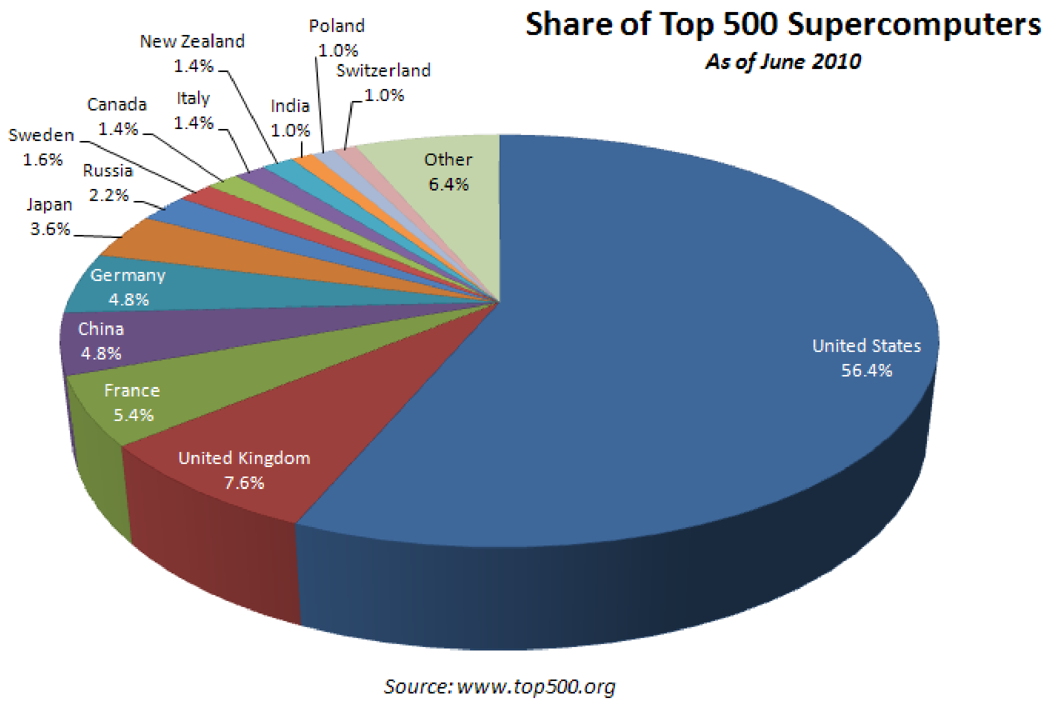
Special-purpose supercomputers are high-performance computing devices with a hardware architecture dedicated to a single problem. This allows the use of specially programmed FPGA chips or even custom VLSI chips, allowing higher price/performance ratios by sacrificing generality. They are used for applications such as astrophysics computation and brute-force codebreaking. Historically a new special-purpose supercomputer has occasionally been faster than the world's fastest general-purpose supercomputer, by some measure. For example, GRAPE-6 was faster than the Earth Simulator in 2002 for a particular special set of problems.

Examples of special-purpose supercomputers:

- Belle, Deep Blue, and Hydra, for playing chess
- Reconfigurable computing machines or parts of machines
- GRAPE, for astrophysics and molecular dynamics
- Deep Crack, for breaking the DES cipher
- MDGRAPE-3, for protein structure computation
- D. E. Shaw Research Anton, for simulating molecular dynamics
- QPACE, for simulations of the strong interaction (Lattice QCD)

The fastest supercomputers today

Measuring supercomputer speed



14 countries account for the vast majority of the world's 500 fastest supercomputers, with over half being located in the United States.

In general, the speed of a supercomputer is measured in "FLOPS" (***FL****oating* ***P****oint* ***O****perations* ***P****er* ***S****econd*), commonly used with an SI prefix such as tera-, combined into the shorthand "TFLOPS" (10^{12} FLOPS, pronounced *teraflops*), or peta-, combined into the shorthand "PFLOPS" (10^{15} FLOPS, pronounced *petaflops*.) This measurement is based on a particular benchmark, which does LU decomposition of a large matrix. This mimics a class of real-world problems, but is significantly easier to compute than a majority of actual real-world problems.

"Petascale" supercomputers can process one quadrillion (10^{15}) (1000 trillion) FLOPS. Exascale is computing performance in the exaflops range. An exaflop is one quintillion (10^{18}) FLOPS (one million teraflops).

The TOP500 list

Since 1993, the fastest supercomputers have been ranked on the TOP500 list according to their LINPACK benchmark results. The list does not claim to be unbiased or definitive, but it is a widely cited current definition of the "fastest" supercomputer available at any given time.

Current fastest supercomputer system

Tianhe-1A is ranked on the TOP500 list as the fastest supercomputer. It consists of 14,336 Intel Xeon CPUs and 7,168 Nvidia Tesla M2050 GPUs with a new interconnect fabric of Chinese origin, reportedly twice the speed of InfiniBand. Tianhe-1A spans 103 cabinets, weighs 155 tons, and consumes 4.04 megawatts of electricity.

Quasi-supercomputing

A Blue Gene/P node card

Some types of large-scale distributed computing for embarrassingly parallel problems take the clustered supercomputing concept to an extreme.

The fastest cluster, Folding@home, reported 9.739 petaflops of processing power as of mid January 2011. Of this, 7.1 petaflops are contributed by clients running on various

GPUs, 1.8 petaflops come from PlayStation 3 systems, and the rest from various computer systems.

Another distributed computing project is the BOINC platform, which hosts a number of distributed computing projects. As of April 2010, BOINC recorded a processing power of over 5 petaflops through over 580,000 active computers on the network. The most active project (measured by computational power), MilkyWay@home, reports processing power of over 1.4 petaflops through over 30,000 active computers.

As of April 2010, GIMPS's distributed Mersenne Prime search currently achieves about 45 teraflops.

Also a “quasi-supercomputer” is Google's search engine system with estimated total processing power of between 126 and 316 teraflops, as of April 2004. In June 2006 the *New York Times* estimated that the Googleplex and its server farms contain 450,000 servers. According to recent estimations, the processing power of Google's cluster might reach from 20 to 100 petaflops.

The PlayStation 3 Gravity Grid uses a network of 16 machines, and exploits the Cell processor for the intended application, which is performing astrophysical simulations of large supermassive black holes capturing smaller compact objects. The Cell processor has a main CPU and 6 floating-point vector processors, giving the machine a net of 16 general-purpose machines and 96 vector processors. This cluster was built in 2007 by Dr. Gaurav Khanna, a professor in the Physics Department of the University of Massachusetts Dartmouth with support from Sony Computer Entertainment and is the first PS3 cluster that generated numerical results that were published in scientific research literature.

Other notable computer clusters are the flash mob cluster and the Beowulf cluster. The flash mob cluster allows the use of any computer in the network, while the Beowulf cluster still requires uniform architecture.

Research and development

IBM is developing the Cyclops64 architecture, intended to create a "supercomputer on a chip".

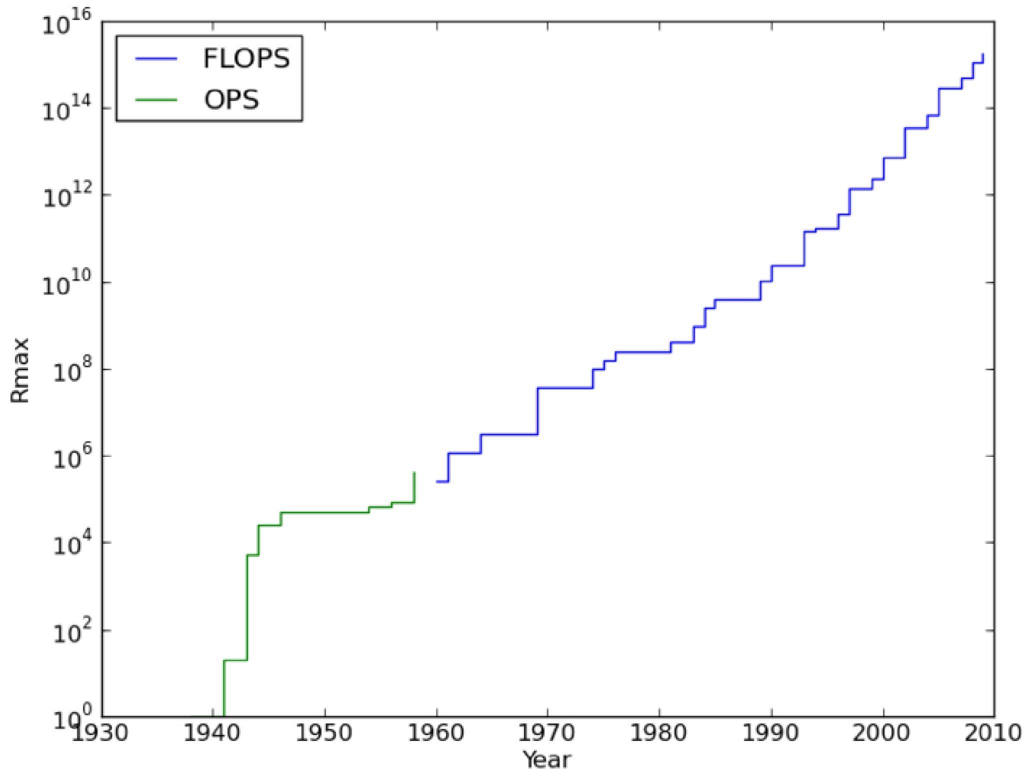
Other PFLOPS projects include one by Narendra Karmarkar in India, a C-DAC effort targeted for 2010, and the Blue Waters Petascale Computing System funded by the NSF ($200 million) that is being built by the NCSA at the University of Illinois at Urbana-Champaign (slated to be completed by 2011).

In May 2008 a collaboration was announced between NASA, SGI and Intel to build a 1 petaflops computer, Pleiades, in 2009, scaling up to 10 PFLOPs by 2012. Meanwhile, IBM is constructing a 20 PFLOPs supercomputer at Lawrence Livermore National Laboratory, named Sequoia, which is scheduled to go online in 2011.

Given the current speed of progress, supercomputers are projected to reach 1 exaflops (10^{18}) (one quintillion FLOPS) in 2019.

Erik P. DeBenedictis of Sandia National Laboratories theorizes that a zettaflops (10^{21}) (one sextillion FLOPS) computer is required to accomplish full weather modeling, which could cover a two week time span accurately. Such systems might be built around 2030.

Timeline of supercomputers



Fastest supercomputers: log speed vs. time

This is a list of the record-holders for fastest general-purpose supercomputer in the world, and the year each one set the record. For entries prior to 1993, this list refers to various sources. From 1993 to present, the list reflects the Top500 listing, and the "Peak speed" is given as the "Rmax" rating.

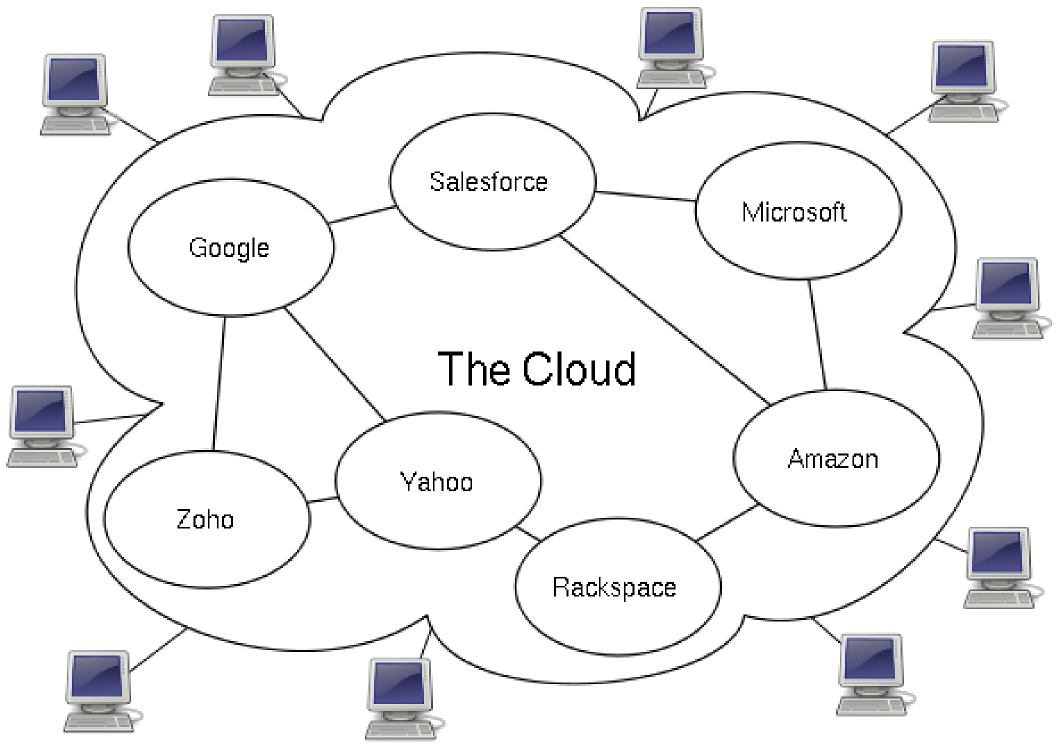
Year	Supercomputer	Peak speed (Rmax)	Location
1938	Zuse Z1	1 OPS	Konrad Zuse, Berlin, Germany
1941	Zuse Z3	20 OPS	Konrad Zuse, Berlin, Germany
1943	Colossus 1	5 kOPS	Post Office Research Station, Bletchley Park, UK

1944	Colossus 2 (Single Processor)	25 kOPS	Post Office Research Station, Bletchley Park, UK
1946	Colossus 2 (Parallel Processor)	50 kOPS	Post Office Research Station, Bletchley Park, UK
1946	UPenn ENIAC (before 1948+ modifications)	5 kOPS	Department of War Aberdeen Proving Ground, Maryland, USA
1954	IBM NORC	67 kOPS	Department of Defense U.S. Naval Proving Ground, Dahlgren, Virginia, USA
1956	MIT TX-0	83 kOPS	Massachusetts Inst. of Technology, Lexington, Massachusetts, USA
1958	IBM AN/FSQ-7	400 kOPS	25 U.S. Air Force sites across the continental USA and 1 site in Canada (52 computers)
1960	UNIVAC LARC	250 kFLOPS	Atomic Energy Commission (AEC) Lawrence Livermore National Laboratory, California, USA
1961	IBM 7030 "Stretch"	1.2 MFLOPS	AEC-Los Alamos National Laboratory, New Mexico, USA
1964	CDC 6600	3 MFLOPS	AEC-Lawrence Livermore National Laboratory, California, USA
1969	CDC 7600	36 MFLOPS	
1974	CDC STAR-100	100 MFLOPS	
1975	Burroughs ILLIAC IV	150 MFLOPS	NASA Ames Research Center, California, USA
1976	Cray-1	250 MFLOPS	Energy Research and Development Administration (ERDA) Los Alamos National Laboratory, New Mexico, USA (80+ sold worldwide)
1981	CDC Cyber 205	400 MFLOPS	(~40 systems worldwide)
1983	Cray X-MP/4	941 MFLOPS	U.S. Department of Energy (DoE) Los Alamos National Laboratory; Lawrence Livermore National Laboratory; Battelle; Boeing
1984	M-13	2.4 GFLOPS	Scientific Research Institute of Computer Complexes, Moscow, USSR
1985	Cray-2/8	3.9 GFLOPS	DoE-Lawrence Livermore National Laboratory, California, USA
1989	ETA10-G/8	10.3 GFLOPS	Florida State University, Florida,

			USA
1990	NEC SX-3/44R	23.2 GFLOPS	NEC Fuchu Plant, Fuchū, Tokyo, Japan
1991	INFN APE 100 (later Quadrics)	100 GFLOPS	INFN, Rome, Italy
1993	Thinking Machines CM-5/1024	59.7 GFLOPS	DoE-Los Alamos National Laboratory; National Security Agency, USA
1993	Fujitsu Numerical Wind Tunnel	124.50 GFLOPS	National Aerospace Laboratory, Tokyo, Japan
1993	Intel Paragon XP/S 140	143.40 GFLOPS	DoE-Sandia National Laboratories, New Mexico, USA
1994	Fujitsu Numerical Wind Tunnel	170.40 GFLOPS	National Aerospace Laboratory, Tokyo, Japan
1996	Hitachi SR2201/1024	220.4 GFLOPS	University of Tokyo, Japan
	Hitachi/Tsukuba CP-PACS/2048	368.2 GFLOPS	Center for Computational Physics, University of Tsukuba, Tsukuba, Japan
1997	Intel ASCI Red/9152	1.338 TFLOPS	DoE-Sandia National Laboratories, New Mexico, USA
1999	Intel ASCI Red/9632	2.3796 TFLOPS	
2000	IBM ASCI White	7.226 TFLOPS	DoE-Lawrence Livermore National Laboratory, California, USA
2002	NEC Earth Simulator	35.86 TFLOPS	Earth Simulator Center, Yokohama, Japan
2004		70.72 TFLOPS	DoE/IBM Rochester, Minnesota, USA
2005	IBM Blue Gene/L	136.8 TFLOPS	DoE/U.S. National Nuclear Security Administration, Lawrence Livermore National Laboratory, California, USA
		280.6 TFLOPS	
2007		478.2 TFLOPS	
2008	IBM Roadrunner	1.026 PFLOPS	DoE-Los Alamos National Laboratory, New Mexico, USA
		1.105 PFLOPS	
2009	Cray Jaguar	1.759 PFLOPS	DoE-Oak Ridge National Laboratory, Tennessee, USA
2010	Tianhe-IA	2.566 PFLOPS	National Supercomputing Center, Tianjin, China

Chapter-11

Cloud Computing



Cloud computing conceptual diagram

Cloud computing refers to the provision of computational resources on demand via a computer network.

Cloud computing can be compared to the supply of electricity and gas, or the provision of telephone, television and postal services. All of these services are presented to the users in a simple way that is easy to understand without the users needing to know how the services are provided. This simplified view is called an abstraction. Similarly, cloud computing offers computer application developers and users an abstract view of services that simplifies and ignores much of the details and inner workings. A provider's offering of abstracted Internet services is often called "The Cloud".

How it works

When a user accesses the cloud for a popular website, many things can happen. The user's IP address, for example, can be used to establish where the user is located (geolocation). DNS services can then direct the user to a cluster of servers that are close to the user so the site can be accessed rapidly and in the user's local language. Users do not log in to a server, but they log in to the service they are using by obtaining a session id or a cookie, which is stored in their browser.

What the user sees in the browser usually comes from a cluster of web servers. The web servers run user interface software which collects commands from the user (mouse clicks, key presses, uploads, etc.) and interprets them. Information is then stored on or retrieved from the database servers or file servers and an updated page is displayed to the user. The data across the multiple servers is synchronised around the world for rapid global access.

Technical description

Cloud computing is computation, software, data access, and storage services that do not require end-user knowledge of the physical location and configuration of the system that delivers the services. Parallels to this concept can be drawn with the electricity grid where end-users consume power resources without any necessary understanding of the component devices in the grid required to provide the service.

Cloud computing describes a new supplement, consumption, and delivery model for IT services based on Internet protocols, and it typically involves provisioning of dynamically scalable and often virtualized resources. It is a byproduct and consequence of the ease-of-access to remote computing sites provided by the Internet. This frequently takes the form of web-based tools or applications that users can access and use through a web browser as if they were programs installed locally on their own computers.

The National Institute of Standards and Technology (NIST) provides a somewhat more objective and specific definition:

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Typical cloud computing providers deliver common business applications online that are accessed from another Web service or software like a Web browser, while the software and data are stored on servers.

Most cloud computing infrastructures consist of services delivered through common centers and built-on servers. Clouds often appear as single points of access for consumers' computing needs. Commercial offerings are generally expected to meet quality of service (QoS) requirements of customers, and typically include service level agreements (SLAs).

Overview

Comparisons

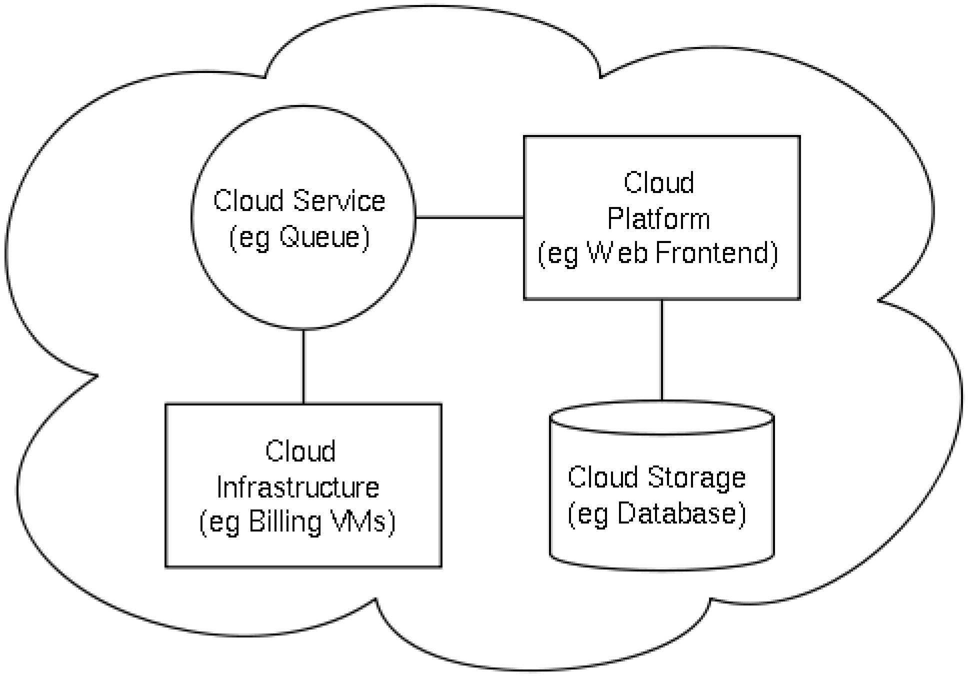
Cloud computing derives characteristics from, but should not be confused with:

1. Autonomic computing — "computer systems capable of self-management."
2. Client–server model – *client–server computing* refers broadly to any distributed application that distinguishes between service providers (servers) and service requesters (clients).
3. Grid computing — "a form of distributed computing and parallel computing, whereby a 'super and virtual computer' is composed of a cluster of networked, loosely coupled computers acting in concert to perform very large tasks."
4. Mainframe computer — powerful computers used mainly by large organizations for critical applications, typically bulk data processing such as census, industry and consumer statistics, enterprise resource planning, and financial transaction processing.
5. Utility computing — the "packaging of computing resources, such as computation and storage, as a metered service similar to a traditional public utility, such as electricity."
6. Peer-to-peer – distributed architecture without the need for central coordination, with participants being at the same time both suppliers and consumers of resources (in contrast to the traditional client–server model).
7. Service-oriented computing – Cloud computing provides services related to computing while, in a reciprocal manner, service-oriented computing consists of the computing techniques that operate on software-as-a-service.

Characteristics

The key characteristic of cloud computing is that the computing is "in the cloud"; that is, the processing (and the related data) is not in a specified, known or static place(s). This is in contrast to a model in which the processing takes place in one or more specific servers that are known. All the other concepts mentioned are supplementary or complementary to this concept.

Architecture



Cloud computing sample architecture

Cloud architecture, the systems architecture of the software systems involved in the delivery of cloud computing, typically involves multiple *cloud components* communicating with each other over application programming interfaces, usually web services and 3-tier architecture. This resembles the Unix philosophy of having multiple programs each doing one thing well and working together over universal interfaces. Complexity is controlled and the resulting systems are more manageable than their monolithic counterparts.

The two most significant components of cloud computing architecture are known as the front end and the back end. The front end is the part seen by the client, i.e. the computer user. This includes the client's network (or computer) and the applications used to access the cloud via a user interface such as a web browser. The back end of the cloud computing architecture is the 'cloud' itself, comprising various computers, servers and data storage devices.

History

The term "cloud" is used as a metaphor for the Internet, based on the cloud drawing used in the past to represent the telephone network, and later to depict the Internet in computer network diagrams as an abstraction of the underlying infrastructure it represents.

Cloud computing is a natural evolution of the widespread adoption of virtualization, service-oriented architecture, autonomic and utility computing. Details are abstracted from end-users, who no longer have need for expertise in, or control over, the technology infrastructure "in the cloud" that supports them.

The underlying concept of cloud computing dates back to the 1960s, when John McCarthy opined that "computation may someday be organized as a public utility." Almost all the modern-day characteristics of cloud computing (elastic provision, provided as a utility, online, illusion of infinite supply), the comparison to the electricity industry and the use of public, private, government and community forms, were thoroughly explored in Douglas Parkhill's 1966 book, *The Challenge of the Computer Utility*.

The actual term "cloud" borrows from telephony in that telecommunications companies, who until the 1990s primarily offered dedicated point-to-point data circuits, began offering Virtual Private Network (VPN) services with comparable quality of service but at a much lower cost. By switching traffic to balance utilization as they saw fit, they were able to utilize their overall network bandwidth more effectively. The cloud symbol was used to denote the demarcation point between that which was the responsibility of the provider, and that which was the responsibility of the user. Cloud computing extends this boundary to cover servers as well as the network infrastructure. The first scholarly use of the term “cloud computing” was in a 1997 lecture by Ramnath Chellappa.

After the dot-com bubble, Amazon played a key role in the development of cloud computing by modernizing their data centers, which, like most computer networks, were using as little as 10% of their capacity at any one time, just to leave room for occasional spikes. Having found that the new cloud architecture resulted in significant internal efficiency improvements whereby small, fast-moving "two-pizza teams" could add new features faster and more easily, Amazon initiated a new product development effort to provide cloud computing to external customers, and launched Amazon Web Service (AWS) on a utility computing basis in 2006.

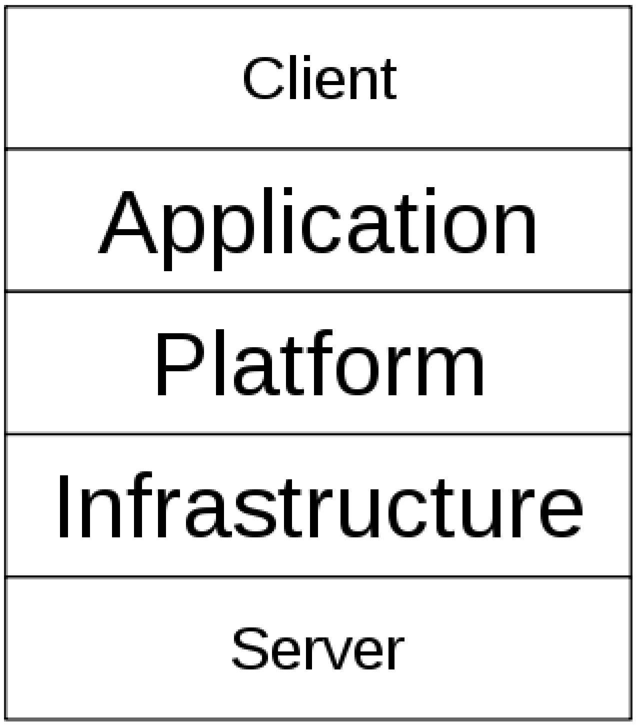
In 2007, Google, IBM and a number of universities embarked on a large-scale cloud computing research project. In early 2008, Eucalyptus became the first open-source, AWS API-compatible platform for deploying private clouds. In early 2008, OpenNebula, enhanced in the RESERVOIR European Commission-funded project, became the first open-source software for deploying private and hybrid clouds, and for the federation of clouds. In the same year, efforts were focused on providing QoS guarantees (as required by real-time interactive applications) to cloud-based infrastructures, in the framework of the IRMOS European Commission-funded project. By mid-2008, Gartner saw an opportunity for cloud computing "to shape the relationship among consumers of IT services, those who use IT services and those who sell them" and observed that "[o]rganisations are switching from company-owned hardware and software assets to per-use service-based models" so that the "projected shift to cloud computing ... will result in dramatic growth in IT products in some areas and significant reductions in other areas."

Key characteristics

- **Agility** improves with users' ability to rapidly and inexpensively re-provision technological infrastructure resources.
- **Application Programming Interface** (API) accessibility to software that enables machines to interact with cloud software in the same way the user interface facilitates interaction between humans and computers. Cloud computing systems typically use REST-based APIs.
- **Cost** is claimed to be greatly reduced and in a public cloud delivery model capital expenditure is converted to operational expenditure. This ostensibly lowers barriers to entry, as infrastructure is typically provided by a third-party and does not need to be purchased for one-time or infrequent intensive computing tasks. Pricing on a utility computing basis is fine-grained with usage-based options and fewer IT skills are required for implementation (in-house).
- **Device and location independence** enable users to access systems using a web browser regardless of their location or what device they are using (e.g., PC, mobile phone). As infrastructure is off-site (typically provided by a third-party) and accessed via the Internet, users can connect from anywhere.
- **Multi-tenancy** enables sharing of resources and costs across a large pool of users thus allowing for:
    - **Centralization** of infrastructure in locations with lower costs (such as real estate, electricity, etc.)
    - **Peak-load capacity** increases (users need not engineer for highest possible load-levels)
    - **Utilization and efficiency** improvements for systems that are often only 10–20% utilized.
- **Reliability** is improved if multiple redundant sites are used, which makes well designed cloud computing suitable for business continuity and disaster recovery.
- **Scalability** via dynamic ("on-demand") provisioning of resources on a fine-grained, self-service basis near real-time, without users having to engineer for peak loads. Performance is monitored, and consistent and loosely coupled architectures are constructed using web services as the system interface.
- **Security** could improve due to centralization of data, increased security-focused resources, etc., but concerns can persist about loss of control over certain sensitive data, and the lack of security for stored kernels. Security is often as good as or better than under traditional systems, in part because providers are able to devote resources to solving security issues that many customers cannot afford. However, the complexity of security is greatly increased when data is distributed over a wider area or greater number of devices and in multi-tenant systems which are being shared by unrelated users. In addition, user access to security audit logs may be difficult or impossible. Private cloud installations are in part motivated by users' desire to retain control over the infrastructure and avoid losing control of information security.
- **Maintenance** of cloud computing applications is easier, because they do not need to be installed on each user's computer. They are easier to support and to improve, as the changes reach the clients instantly.

Layers

Once an Internet Protocol connection is established among several computers, it is possible to share services within any one of the following layers.



Client

A *cloud client* consists of computer hardware and/or computer software that relies on cloud computing for application delivery, or that is specifically designed for delivery of cloud services and that, in either case, is essentially useless without it. Examples include some computers, phones and other devices, operating systems and browsers.

Application

Cloud application services or "Software as a Service (SaaS)" deliver software as a service over the Internet, eliminating the need to install and run the application on the customer's own computers and simplifying maintenance and support. People tend to use the terms "SaaS" and "cloud" interchangeably, when in fact they are two different things. Key characteristics include:

- Network-based access to, and management of, commercially available (i.e., not custom) software
- Activities that are managed from central locations rather than at each customer's site, enabling customers to access applications remotely via the Web

- Application delivery that typically is closer to a one-to-many model (single instance, multi-tenant architecture) than to a one-to-one model, including architecture, pricing, partnering, and management characteristics
- Centralized feature updating, which obviates the need for downloadable patches and upgrades

Platform

Cloud platform services or "Platform as a Service (PaaS)" deliver a computing platform and/or solution stack as a service, often consuming cloud infrastructure and sustaining cloud applications. It facilitates deployment of applications without the cost and complexity of buying and managing the underlying hardware and software layers.

Infrastructure

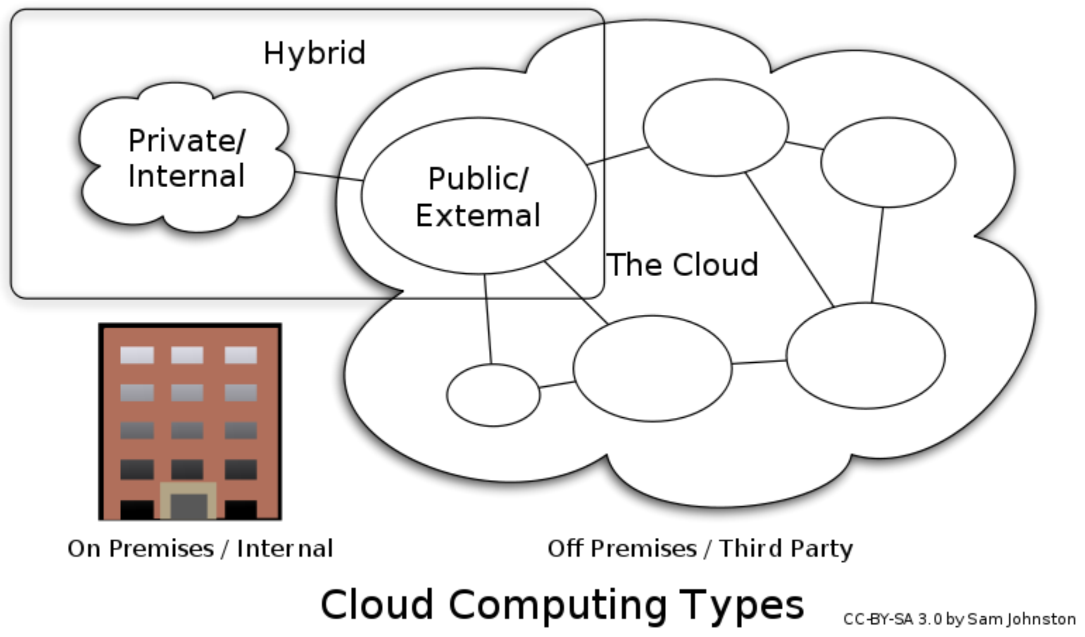
Cloud infrastructure services, also known as "Infrastructure as a Service (IaaS)", delivers computer infrastructure – typically a platform virtualization environment – as a service. Rather than purchasing servers, software, data-center space or network equipment, clients instead buy those resources as a fully outsourced service. Suppliers typically bill such services on a utility computing basis and amount of resources consumed (and therefore the cost) will typically reflect the level of activity. IaaS evolved from virtual private server offerings.

Cloud infrastructure often takes the form of a tier 3 data center with many tier 4 attributes, assembled from hundreds of virtual machines.

Server

The servers layer consists of computer hardware and/or computer software products that are specifically designed for the delivery of cloud services, including multi-core processors, cloud-specific operating systems and combined offerings.

Deployment models



Cloud computing types

Public cloud

Public cloud or external cloud describes cloud computing in the traditional mainstream sense, whereby resources are dynamically provisioned on a fine-grained, self-service basis over the Internet, via web applications/web services, from an off-site third-party provider who bills on a fine-grained utility computing basis.

Community cloud

A community cloud may be established where several organizations have similar requirements and seek to share infrastructure so as to realize some of the benefits of cloud computing. With the costs spread over fewer users than a public cloud (but more than a single tenant) this option is more expensive but may offer a higher level of privacy, security and/or policy compliance. Examples of community clouds include Google's "Gov Cloud".

Hybrid cloud and hybrid IT delivery

The main responsibility of the IT department is to deliver services to the business. With the proliferation of cloud computing (both private and public) and the fact that IT departments must also deliver services via traditional, in-house methods, the newest catch-phrase has become “hybrid cloud computing.” Hybrid cloud is also called hybrid delivery by the major vendors including HP, IBM, Oracle and VMware who offer technology to manage the complexity in managing the performance, security and privacy concerns that results from the mixed delivery methods of IT services.

A hybrid storage cloud uses a combination of public and private storage clouds. Hybrid storage clouds are often useful for archiving and backup functions, allowing local data to be replicated to a public cloud.

Another perspective on deploying a web application in the cloud is using Hybrid Web Hosting, where the hosting infrastructure is a mix between cloud hosting and managed dedicated servers – this is most commonly achieved as part of a web cluster in which some of the nodes are running on real physical hardware and some are running on cloud server instances.

Combined cloud

Two clouds that have been joined together are more correctly called a "combined cloud". A combined cloud environment consisting of multiple internal and/or external providers "will be typical for most enterprises". By integrating multiple cloud services users may be able to ease the transition to *public cloud* services while avoiding issues such as PCI compliance.

Private cloud

Douglas Parkhill first described the concept of a "private computer utility" in his 1966 book *The Challenge of the Computer Utility*. The idea was based upon direct comparison with other industries (e.g. the electricity industry) and the extensive use of hybrid supply models to balance and mitigate risks.

"Private cloud" and "internal cloud" have been described as neologisms, but the concepts themselves pre-date the term cloud by 40 years. Even within modern utility industries, hybrid models still exist despite the formation of reasonably well-functioning markets and the ability to combine multiple providers.

Some vendors have used the terms to describe offerings that emulate cloud computing on private networks. These (typically virtualization automation) products offer the ability to host applications or virtual machines in a company's own set of hosts. These provide the benefits of utility computing – shared hardware costs, the ability to recover from failure, and the ability to scale up or down depending upon demand.

Private clouds have attracted criticism because users "still have to buy, build, and manage them" and thus do not benefit from lower up-front capital costs and less hands-on management, essentially "[lacking] the economic model that makes cloud computing such an intriguing concept". Enterprise IT organizations use their own private cloud(s) for mission critical and other operational systems to protect critical infrastructures.

Cloud engineering

Cloud engineering is the application of a systematic, disciplined, quantifiable, and interdisciplinary approach to the ideation, conceptualization, development, operation, and maintenance of cloud computing, as well as the study and applied research of the approach, i.e., the application of engineering to cloud. It is a maturing and evolving discipline to facilitate the adoption, strategization, operationalization, industrialization, standardization, productization, commoditization, and governance of cloud solutions, leading towards a cloud ecosystem. Cloud engineering is also known as cloud service engineering.

Cloud storage

Cloud storage is a model of networked computer data storage where data is stored on multiple virtual servers, generally hosted by third parties, rather than being hosted on dedicated servers. Hosting companies operate large data centers; and people who require their data to be hosted buy or lease storage capacity from them and use it for their storage needs. The data center operators, in the background, virtualize the resources according to the requirements of the customer and expose them as virtual servers, which the customers can themselves manage. Physically, the resource may span across multiple servers.

The Intercloud

The Intercloud is an interconnected global "cloud of clouds" and an extension of the Internet "network of networks" on which it is based.[8] The term was first used in the context of cloud computing in 2007 when Kevin Kelly stated that "eventually we'll have the intercloud, the cloud of clouds. This Intercloud will have the dimensions of one machine comprising all servers and attendant cloudbooks on the planet.". It became popular in 2009 and has also been used to describe the datacenter of the future.

The Intercloud scenario is based on the key concept that each single cloud does not have infinite physical resources. If a cloud saturates the computational and storage resources of its virtualization infrastructure, it could not be able to satisfy further requests for service allocations sent from its clients. The Intercloud scenario aims to address such situation, and in theory, each cloud can use the computational and storage resources of the virtualization infrastructures of other clouds. Such form of pay-for-use may introduce new business opportunities among cloud providers if they manage to go beyond theoretical framework. Nevertheless, the Intercloud raises many more challenges than solutions concerning cloud federation, security, interoperability, quality of service, vendor's lock-ins, trust, legal issues, monitoring and billing.

The concept of a competitive utility computing market which combined many computer utilities together was originally described by Douglas Parkhill in his 1966 book, the "Challenge of the Computer Utility". This concept has been subsequently used many times over the last 40 years and is identical to the Intercloud.

Issues

Privacy

The cloud model has been criticized by privacy advocates for the greater ease in which the companies hosting the cloud services control, and thus, can monitor at will, lawfully or unlawfully, the communication and data stored between the user and the host company. Instances such as the secret NSA program, working with AT&T, and Verizon, which recorded over 10 million phone calls between American citizens, causes uncertainty among privacy advocates, and the greater powers it gives to telecommunication companies to monitor user activity. While there have been efforts (such as US-EU Safe Harbor) to "harmonize" the legal environment, providers such as Amazon still cater to major markets (typically the United States and the European Union) by deploying local infrastructure and allowing customers to select "availability zones."

Compliance

In order to obtain compliance with regulations including FISMA, HIPAA and SOX in the United States, the Data Protection Directive in the EU and the credit card industry's PCI DSS, users may have to adopt *community* or *hybrid* deployment modes which are typically more expensive and may offer restricted benefits. This is how Google is able to "manage and meet additional government policy requirements beyond FISMA" and Rackspace Cloud are able to claim PCI compliance. Customers in the EU contracting with cloud providers established outside the EU/EEA have to adhere to the EU regulations on export of personal data.

Many providers also obtain SAS 70 Type II certification but this has been criticised on the grounds that the hand-picked set of goals and standards determined by the auditor and the auditee are often not disclosed and can vary widely. Providers typically make this information available on request, under non-disclosure agreement.

Legal

In March 2007, Dell applied to trademark the term "cloud computing" (U.S. Trademark 77,139,082) in the United States. The "Notice of Allowance" the company received in July 2008 was canceled in August, resulting in a formal rejection of the trademark application less than a week later. Since 2007, the number of trademark filings covering cloud computing brands, goods and services has increased at an almost exponential rate. As companies sought to better position themselves for cloud computing branding and marketing efforts, cloud computing trademark filings increased by 483% between 2008 and 2009. In 2009, 116 cloud computing trademarks were filed, and trademark analysts predict that over 500 such marks could be filed during 2010.

Other legal cases may shape the use of cloud computing by the public sector. On October 29, 2010, Google filed a lawsuit against the U.S. Department of Interior, which opened up a bid for software that required that bidders use Microsoft's Business Productivity

Online Suite. Google sued, calling the requirement "unduly restrictive of competition." Scholars have pointed out that, beginning in 2005, the prevalence of open standards and open source may have an impact on the way that public entities choose to select vendors.

Open source

Open source software has provided the foundation for many cloud computing implementations. In November 2007, the Free Software Foundation released the Affero General Public License, a version of GPLv3 intended to close a perceived legal loophole associated with free software designed to be run over a network.

Open standards

Most cloud providers expose APIs which are typically well-documented (often under a Creative Commons license) but also unique to their implementation and thus not interoperable. Some vendors have adopted others' APIs and there are a number of open standards under development, including the OGF's Open Cloud Computing Interface. The Open Cloud Consortium (OCC) is working to develop consensus on early cloud computing standards and practices.

Security

The relative security of cloud computing services is a contentious issue which may be delaying its adoption. Issues barring the adoption of cloud computing are due in large part to the private and public sectors unease surrounding the external management of security based services. It is the very nature of cloud computing based services, private or public, that promote external management of provided services. This delivers great incentive amongst cloud computing service providers in producing a priority in building and maintaining strong management of secure services.

Organizations have been formed in order to provide standards for a better future in cloud computing services. One organization in particular, the Cloud Security Alliance is a non-profit organization formed to promote the use of best practices for providing security assurance within cloud computing.

Availability and performance

In addition to concerns about security, businesses are also worried about acceptable levels of availability and performance of applications hosted in the cloud.

There are also concerns about a cloud provider shutting down for financial or legal reasons, which has happened in a number of cases.

Sustainability and siting

Although cloud computing is often assumed to be a form of "green computing", there is as of yet no published study to substantiate this assumption. Siting the servers affects the environmental effects of cloud computing. In areas where climate favors natural cooling and renewable electricity is readily available, the environmental effects will be more moderate. Thus countries with favorable conditions, such as Finland, Sweden and Switzerland, are trying to attract cloud computing data centers.

SmartBay, marine research infrastructure of sensors and computational technology, is being developed using cloud computing, an emerging approach to shared infrastructure in which large pools of systems are linked together to provide IT services.

Research

A number of universities, vendors and government organizations are investing in research around the topic of cloud computing. Academic institutions include University of Melbourne (Australia), Georgia Tech, Yale, Wayne State, Virginia Tech, University of Wisconsin–Madison, Carnegie Mellon, MIT, Indiana University, University of Massachusetts, University of Maryland, IIT Bombay, North Carolina State University, Purdue University, University of California, University of Washington, University of Virginia, University of Utah, University of Minnesota, among others.

Joint government, academic and vendor collaborative research projects include the IBM/Google Academic Cloud Computing Initiative (ACCI). In October 2007 IBM and Google announced the multi- university project designed to enhance students' technical knowledge to address the challenges of cloud computing. In April 2009, the National Science Foundation joined the ACCI and awarded approximately $5 million in grants to 14 academic institutions.

In July 2008, HP, Intel Corporation and Yahoo! announced the creation of a global, multi-data center, open source test bed, called Open Cirrus, designed to encourage research into all aspects of cloud computing, service and data center management. Open Cirrus partners include the NSF, the University of Illinois (UIUC), Karlsruhe Institute of Technology, the Infocomm Development Authority (IDA) of Singapore, the Electronics and Telecommunications Research Institute (ETRI) in Korea, the Malaysian Institute for Microelectronic Systems(MIMOS), and the Institute for System Programming at the Russian Academy of Sciences (ISPRAS). In Sept. 2010, more researchers joined the HP/Intel/Yahoo Open Cirrus project for cloud computing research. The new researchers are China Mobile Research Institute (CMRI), Spain's Supercomputing Center of Galicia (CESGA by its Spanish acronym), Georgia Tech's Center for Experimental Research in Computer Systems (CERCS) and China Telecom.

In July 2010, HP Labs India announced a new cloud-based technology designed to simplify taking content and making it mobile-enabled, even from low-end devices. Called SiteonMobile, the new technology is designed for emerging markets where people are more likely to access the internet via mobile phones rather than computers. In Nov. 2010, HP formally opened its Government Cloud Theatre, located at the HP Labs site in Bristol, England. The demonstration facility highlights high-security, highly flexible cloud computing based on intellectual property developed at HP Labs. The aim of the facility is to lessen fears about the security of the cloud. HP Labs Bristol is HP’s second-largest central research location and currently is responsible for researching cloud computing and security.

The IEEE Technical Committee on Services Computing in IEEE Computer Society sponsors the IEEE International Conference on Cloud Computing (CLOUD). CLOUD 2010 was held on July 5–10, 2010 in Miami, Florida

On March 23, 2011, Google, Microsoft, HP, Yahoo, Verizon, Deutsche Telecom and 17 other companies formed a nonprofit organization called Open Networking Foundation, focused on providing support for a new cloud initiative called Software-Defined Networking. The initiative is meant to speed innovation through simple software changes in telecommunications networks, wireless networks, data centers and other networking areas.

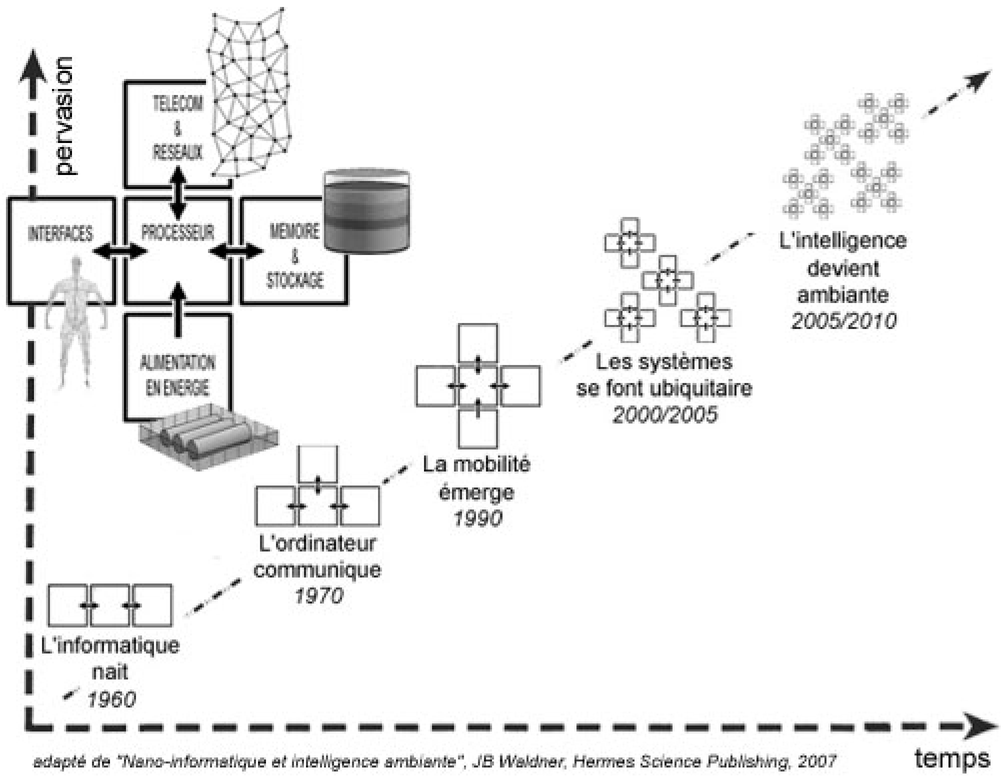
Criticism of the term

Some have come to criticize the term as being either too unspecific or even misleading. CEO Larry Ellison of Oracle Corporation asserts that cloud computing is "everything that we already do", claiming that the company could simply "change the wording on some of our ads" to deploy their cloud-based services. Forrester Research VP Frank Gillett questions the very nature of and motivation behind the push for cloud computing, describing what he calls "cloud washing" in the industry whereby companies relabel their products as cloud computing resulting in a lot of marketing innovation on top of real innovation. GNU's Richard Stallman insists that the industry will only use the model to deliver services at ever increasing rates over proprietary systems, otherwise likening it to a "marketing hype campaign".

Chapter-12

Ambient Intelligence and Ubiquitous Computing

Ambient intelligence


pervasion
TELECOM & RESEAUX
INTERFACES
PROCESSEUR
MEMOIRE & STOCKAGE
ALIMENTATION EN ENERGIE
L'informatique nait 1960
L'ordinateur communique 1970
La mobilité émerge 1990
Les systèmes se font ubiquitaire 2000/2005
L'intelligence devient ambiante 2005/2010
temps
adapté de "Nano-informatique et intelligence ambiante", JB Waldner, Hermes Science Publishing, 2007

An (expected) evolution of computing from 1960–2010.

In computing, **ambient intelligence** (`AmI`) refers to electronic environments that are sensitive and responsive to the presence of people. Ambient intelligence is a vision on the future of consumer electronics, telecommunications and computing that was originally developed in the late 1990s for the time frame 2010–2020. In an ambient intelligence world, devices work in concert to support people in carrying out their everyday life activities, tasks and rituals in easy, natural way using information and intelligence that is hidden in the network connecting these devices. As these devices grow smaller, more connected and more integrated into our environment, the technology disappears into our surroundings until only the user interface remains perceivable by users.

The ambient intelligence paradigm builds upon pervasive computing, ubiquitous computing, profiling practices, context awareness, and human-centric computer interaction design and is characterized by systems and technologies that are (Zelkha & Epstein 1998; Aarts, Harwig & Schuurmans 2001):

- embedded: many networked devices are integrated into the environment
- context aware: these devices can recognize you and your situational context
- personalized: they can be tailored to your needs
- adaptive: they can change in response to you
- anticipatory: they can anticipate your desires without conscious mediation.

Ambient intelligence is closely related to the long term vision of an intelligent service system in which technologies are able to automate a platform embedding the required devices for powering context aware, personalized, adaptive and anticipatory services.

A typical context of ambient intelligence environment is a Home environment (Bieliková & Krajcovic 2001).

Overview

More and more people make decisions based on the effect their actions will have on their own inner, mental world. This experience-driven way of acting is a change from the past when people were primarily concerned about the use value of products and services, and is the basis for the experience economy. Ambient intelligence addresses this shift in existential view by emphasizing people and user experience.

The interest in user experience also grew in importance in the late 1990s because of the overload of products and services in the information society that were difficult to understand and hard to use. A strong call emerged to design things from a user's point of view. Ambient intelligence is influenced by user-centered design where the user is placed in the center of the design activity and asked to give feedback through specific user evaluations and tests to improve the design or even co-create the design together with the designer (participatory design) or with other users (end-user development).

In order for AmI to become a reality a number of key technologies are required:

- Unobtrusive hardware (Miniaturization, Nanotechnology, smart devices, sensors etc.)

- Seamless mobile/fixed communication and computing infrastructure (interoperability, wired and wireless networks, service-oriented architecture, semantic web etc.)
- Dynamic and massively distributed device networks, which are easy to control and program (e.g. service discovery, auto-configuration, end-user programmable devices and systems etc.).
- Human-centric computer interfaces (intelligent agents, multimodal interaction, context awareness etc.)
- Dependable and secure systems and devices (self-testing and self repairing software, privacy ensuring technology etc.)

History

In 1998, the board of management of Philips commissioned a series of presentations and internal workshops, organized by Eli Zelkha and Brian Epstein of Palo Alto Ventures (who, with Simon Birrell, coined the name 'Ambient Intelligence') to investigate different scenarios that would transform the high-volume consumer electronic industry from the current "fragmented with features" world into a world in 2020 where user-friendly devices support ubiquitous information, communication and entertainment. While developing the Ambient Intelligence concept, Palo Alto Ventures created the keynote address for Roel Pieper of Philips for the Digital Living Room Conference, 1998. The group included Eli Zelkha, Brian Epstein, Simon Birrell, Doug Randall, and Clark Dodsworth. In the years after, these developments grew more mature. In 1999, Philips joined the Oxygen alliance, an international consortium of industrial partners within the context of the MIT Oxygen project , aimed at developing technology for the computer of the 21st century. In 2000, plans were made to construct a feasibility and usability facility dedicated to Ambient Intelligence. This HomeLab officially opened on 24 April 2002.

Along with the development of the vision at Philips, a number of parallel initiatives started to explore ambient intelligence in more detail. Following the advice of the Information Society and Technology Advisory Group (ISTAG), the European Commission used the vision for the launch of their sixth framework (FP6) in Information, Society and Technology (IST), with a subsidiary budget of 3.7 billion euros. The European Commission played a crucial role in the further development of the AmI vision. As a result of many initiatives the AmI vision gained traction. During the past few years several major initiatives have been started. Fraunhofer Society started several activities in a variety of domains including multimedia, microsystems design and augmented spaces. MIT started an Ambient Intelligence research group at their Media Lab . Several more research projects started in a variety of countries such as USA, Canada, Spain, France and the Netherlands. In 2004, the first European symposium on Ambient Intelligence (EUSAI) was held and many other conferences have been held that address special topics in AmI.

Example scenario

Ellen returns home after a long day's work. At the front door she is recognized by an intelligent surveillance camera, the door alarm is switched off, and the door unlocks and opens. When she enters the hall the house map indicates that her husband Peter is at an art fair in Paris, and that her daughter Charlotte is in the children's playroom, where she is playing with an interactive screen. The remote children surveillance service is notified that she is at home, and subsequently the on-line connection is switched off. When she enters the kitchen the family memo frame lights up to indicate that there are new messages. The shopping list that has been composed needs confirmation before it is sent to the supermarket for delivery. There is also a message notifying that the home information system has found new information on the semantic Web about economic holiday cottages with sea sight in Spain. She briefly connects to the playroom to say hello to Charlotte, and her video picture automatically appears on the flat screen that is currently used by Charlotte. Next, she connects to Peter at the art fair in Paris. He shows her through his contact lens camera some of the sculptures he intends to buy, and she confirms his choice. In the mean time she selects one of the displayed menus that indicate what can be prepared with the food that is currently available from the pantry and the refrigerator. Next, she switches to the video on demand channel to watch the latest news program. Through the follow me she switches over to the flat screen in the bedroom where she is going to have her personalized workout session. Later that evening, after Peter has returned home, they are chatting with a friend in the living room with their personalized ambient lighting switched on. They watch the virtual presenter that informs them about the programs and the information that have been recorded by the home storage server earlier that day.

Criticism

The Ambient intelligence vision is not without criticism. Its immersive, personalized, context-aware and anticipatory characteristics bring up societal, political and cultural concerns about the loss of consumer privacy, power concentration in large organizations, fear for an increasingly individualized, fragmented society and hyperreal environments where the virtual is indistinguishable from the real (hyperreality). Several research groups and communities are investigating the social-economical, political and cultural aspects of ambient intelligence. New thinking on Ambient Intelligence distances itself therefore from some of the original characteristics such as adaptive and anticipatory behaviour and emphasizes empowerment and participation to place control in the hands of people instead of organizations.

Social and political aspects

The ISTAG advisory group suggests that the following characteristics will permit the societal acceptance of ambient intelligence:

- AmI should facilitate human contact.
- AmI should be oriented towards community and cultural enhancement.

- AmI should help to build knowledge and skills for work, better quality of work, citizenship and consumer choice.
- AmI should inspire trust and confidence.
- AmI should be consistent with long term sustainability — personal, societal and environmental — and with life-long learning.
- AmI should be made easy to live with and controllable by ordinary people.

Business models

The ISTAG group acknowledges the following entry points to AmI business landscape:

- Initial premium value niche markets in industrial, commercial or public applications where enhanced interfaces are needed to support human performance in fast moving or delicate situations.
- Start-up and spin-off opportunities from identifying potential service requirements and putting the services together that meet these new needs.
- High access-low entry cost based on a loss leadership model in order to create economies of scale (mass customization).
- Audience or customer's attention economy as a basis for 'free' end-user services paid for by advertising or complementary services or goods.
- Self-provision – based upon the network economies of very large user communities providing information as a gift or at near zero cost.

Technologies

A variety of technologies can be used to enable Ambient intelligence environments such as (Gasson & Warwick 2007):

- RFID
- Ict implant
- Sensors
- Software agents
- Affective computing
- Nanotechnology
- Biometrics

- The Diamond Age by Neal Stephenson. The Diamond Age depicts a world completely changed by the full development of nanotechnology that is present everywhere.

Ubiquitous computing

Ubiquitous computing (**ubicomp**) is a post-desktop model of human-computer interaction in which information processing has been thoroughly integrated into everyday objects and activities. In the course of ordinary activities, someone "using" ubiquitous computing engages many computational devices and systems simultaneously, and may not necessarily even be aware that they are doing so. This model is usually considered an advancement from the desktop paradigm. More formally Ubiquitous computing is defined as "machines that fit the human environment instead of forcing humans to enter theirs."

This paradigm is also described as **pervasive computing**, ambient intelligence., where each term emphasizes slightly different aspects. When primarily concerning the objects involved, it is also **physical computing**, the *Internet of Things*, *haptic computing*, and *things that think*. Rather than propose a single definition for ubiquitous computing and for these related terms, a taxonomy of properties for ubiquitous computing has been proposed, from which different kinds or flavors of ubiquitous systems and applications can be described.

Core concept

At their core, all models of ubiquitous computing (also called pervasive computing) share a vision of small, inexpensive, robust networked processing devices, distributed at all scales throughout everyday life and generally turned to distinctly common-place ends. For example, a domestic ubiquitous computing environment might interconnect lighting and environmental controls with personal biometric monitors woven into clothing so that illumination and heating conditions in a room might be modulated, continuously and imperceptibly. Another common scenario posits refrigerators "aware" of their suitably-tagged contents, able to both plan a variety of menus from the food actually on hand, and warn users of stale or spoiled food.

Ubiquitous computing presents challenges across computer science: in systems design and engineering, in systems modelling, and in user interface design. Contemporary human-computer interaction models, whether command-line, menu-driven, or GUI-based, are inappropriate and inadequate to the ubiquitous case. This suggests that the "natural" interaction paradigm appropriate to a fully robust ubiquitous computing has yet to emerge - although there is also recognition in the field that in many ways we are already living in an ubicomp world. Contemporary devices that lend some support to this latter idea include mobile phones, digital audio players, radio-frequency identification tags, GPS, and interactive whiteboards.

Mark Weiser proposed three basic forms for ubiquitous system devices**.**

Tabs: wearable centimetre sized devices

- *Pads*: hand-held decimetre-sized devices
- *Boards*: metre sized interactive display devices.

These three forms proposed by Weiser are characterized by being macro-sized, having a planar form and on incorporating visual output displays. If we relax each of these three characteristics we can expand this range into a much more diverse and potentially more useful range of Ubiquitous Computing devices. Hence, three additional forms for ubiquitous systems have been proposed:

- *Dust*: miniaturized devices can be without visual output displays, e.g., Micro Electro-Mechanical Systems (MEMS), ranging from nanometres through micrometers to millimetres.
- *Skin*: fabrics based upon light emitting and conductive polymers, organic computer devices, can be formed into more flexible non-planar display surfaces and products such as clothes and curtains. MEMS device can also be painted onto various surfaces so that a variety of physical world structures can act as networked surfaces of MEMS.
- *Clay*: ensembles of MEMS can be formed into arbitrary three dimensional shapes as artefacts resembling many different kinds of physical object.

In his book *The Rise of the Network Society*, Manuel Castells suggests that there is an ongoing shift from already-decentralised, stand-alone microcomputers and mainframes towards entirely pervasive computing. In his model of a pervasive computing system, Castells uses the example of the Internet as the start of a pervasive computing system. The logical progression from that paradigm is a system where that networking logic becomes applicable in every realm of daily activity, in every location and every context. Castells envisages a system where billions of miniature, ubiquitous inter-communication devices will be spread worldwide, "like pigment in the wall paint".

History

Mark Weiser coined the phrase "ubiquitous computing" around 1988, during his tenure as Chief Technologist of the Xerox Palo Alto Research Center (PARC). Both alone and with PARC Director and Chief Scientist John Seely Brown, Weiser wrote some of the earliest papers on the subject, largely defining it and sketching out its major concerns.

Recognizing that the extension of processing power into everyday scenarios would necessitate understandings of social, cultural and psychological phenomena beyond its proper ambit, Weiser was influenced by many fields outside computer science, including "philosophy, phenomenology, anthropology, psychology, post-Modernism, sociology of science and feminist criticism." He was explicit about "the humanistic origins of the 'invisible ideal in post-modernist thought'", referencing as well the ironically dystopian Philip K. Dick novel Ubik.

MIT has also contributed significant research in this field, notably Hiroshi Ishii's *Things That Think* consortium at the Media Lab and the CSAIL effort known as Project Oxygen.

Other major contributors include Georgia Tech's College of Computing, NYU's Interactive Telecommunications Program, UC Irvine's Department of Informatics, Microsoft Research, Intel Research and Equator, Ajou University UCRi & CUS.

Examples

One of the earliest ubiquitous systems was artist Natalie Jeremijenko's "Live Wire", also known as "Dangling String", installed at Xerox PARC during Mark Weiser's time there. This was a piece of string attached to a stepper motor and controlled by a LAN connection; network activity caused the string to twitch, yielding a *peripherally noticeable* indication of traffic. Weiser called this an example of *calm technology*.

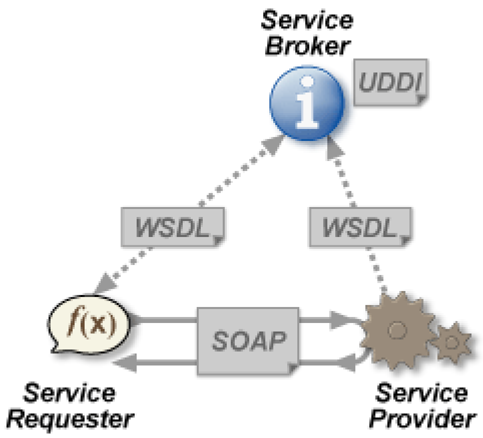
Ambient Devices has produced an "orb", a "dashboard", and a "weather beacon": these decorative devices receive data from a wireless network and report current events, such as stock prices and the weather, like the Nabaztag produced by Violet Snowden.

Current research

Ubiquitous computing touches on a wide range of research topics, including distributed computing, mobile computing, sensor networks, human-computer interaction, and artificial intelligence.

Chapter-13

Web Service



Web services architecture.

A **Web service** is a method of communication between two electronic devices over a network.

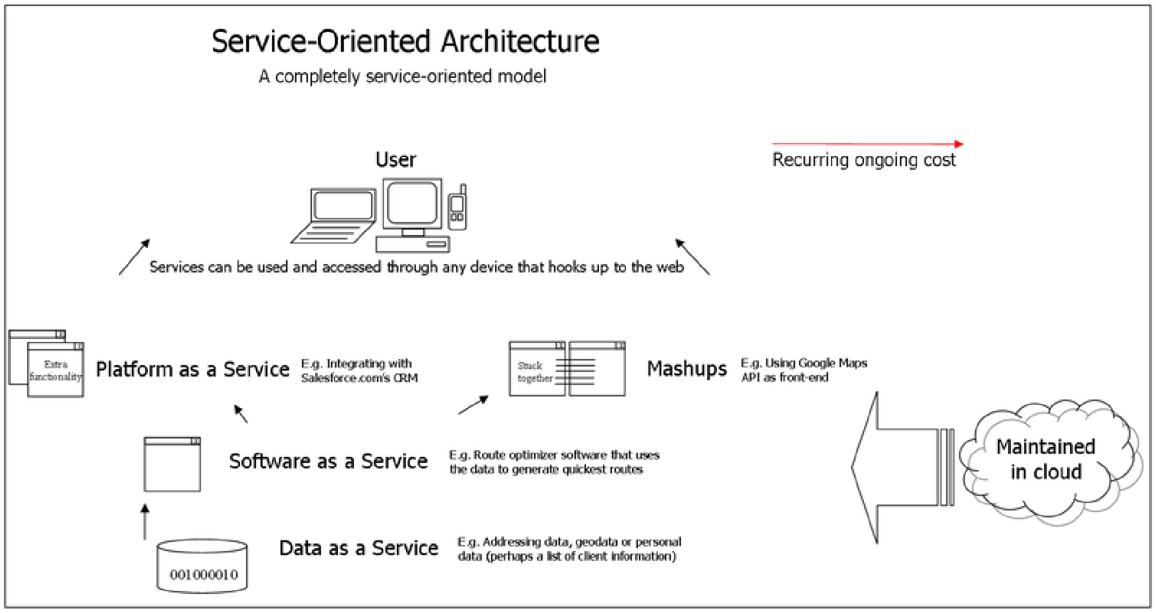
The W3C defines a "Web service" as "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

The W3C also states, "We can identify two major classes of Web services, REST-compliant Web services, in which the primary purpose of the service is to manipulate XML representations of Web resources using a uniform set of "stateless" operations; and arbitrary Web services, in which the service may expose an arbitrary set of operations."

Big Web services

"Big Web services" use Extensible Markup Language (XML) messages that follow the SOAP standard and have been popular with traditional enterprises. In such systems, there is often a machine-readable description of the operations offered by the service written in the Web Services Description Language (WSDL). The latter is not a requirement of a SOAP *endpoint*, but it is a prerequisite for automated client-side code generation in many Java and .NET SOAP frameworks (frameworks such as Apache Axis2, Apache CXF, and Spring being notable exceptions). Some industry organizations, such as the WS-I, mandate both SOAP and WSDL in their definition of a Web service.

Web API


Service-Oriented Architecture
A completely service-oriented model
User
Recurring ongoing cost
Services can be used and accessed through any device that hooks up to the web
Extra functionality
Platform as a Service
E.g. Integrating with Salesforce.com's CRM
Stuck together
Mashups
E.g. Using Google Maps API as front-end
Maintained in cloud
Software as a Service
E.g. Route optimizer software that uses the data to generate quickest routes
Data as a Service
E.g. Addressing data, geodata or personal data (perhaps a list of client information)
001000010

Web services in a service-oriented architecture.

Web API is a development in Web services (in a movement called Web 2.0) where emphasis has been moving away from SOAP based services towards Representational State Transfer (REST) based communications. REST services do not require XML, SOAP, or WSDL service-API definitions.

Web APIs allow the combination of multiple Web services into new applications known as mashups.

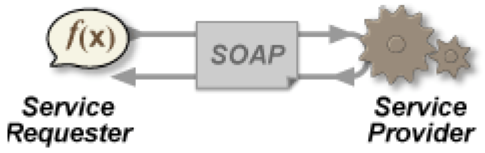
When used in the context of Web development, Web API is typically a defined set of Hypertext Transfer Protocol (HTTP) request messages along with a definition of the structure of response messages, usually expressed in an Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format.

When running composite Web services, each sub service can be considered autonomous. The user has no control over these services. Also the Web services themselves are not reliable; the service provider may remove, change or update their services without giving notice to users. The reliability and fault tolerance is not well supported; faults may happen during the execution. Exception handling in the context of Web services is still an open research issue. Still it can be handled by responding with an error object to the client.

Styles of use

Web services are a set of tools that can be used in a number of ways. The three most common styles of use are RPC, SOAP and REST.

Remote procedure calls



Architectural elements involved in the XML-RPC.

RPC Web services present a distributed function (or method) call interface that is familiar to many developers. Typically, the basic unit of RPC Web services is the WSDL operation.

The first Web services tools were focused on RPC, and as a result this style is widely deployed and supported. However, it is sometimes criticized for not being loosely coupled, because it was often implemented by mapping services directly to language-specific functions or method calls. Many vendors felt this approach to be a dead end, and pushed for RPC to be disallowed in the WS-I Basic Profile.

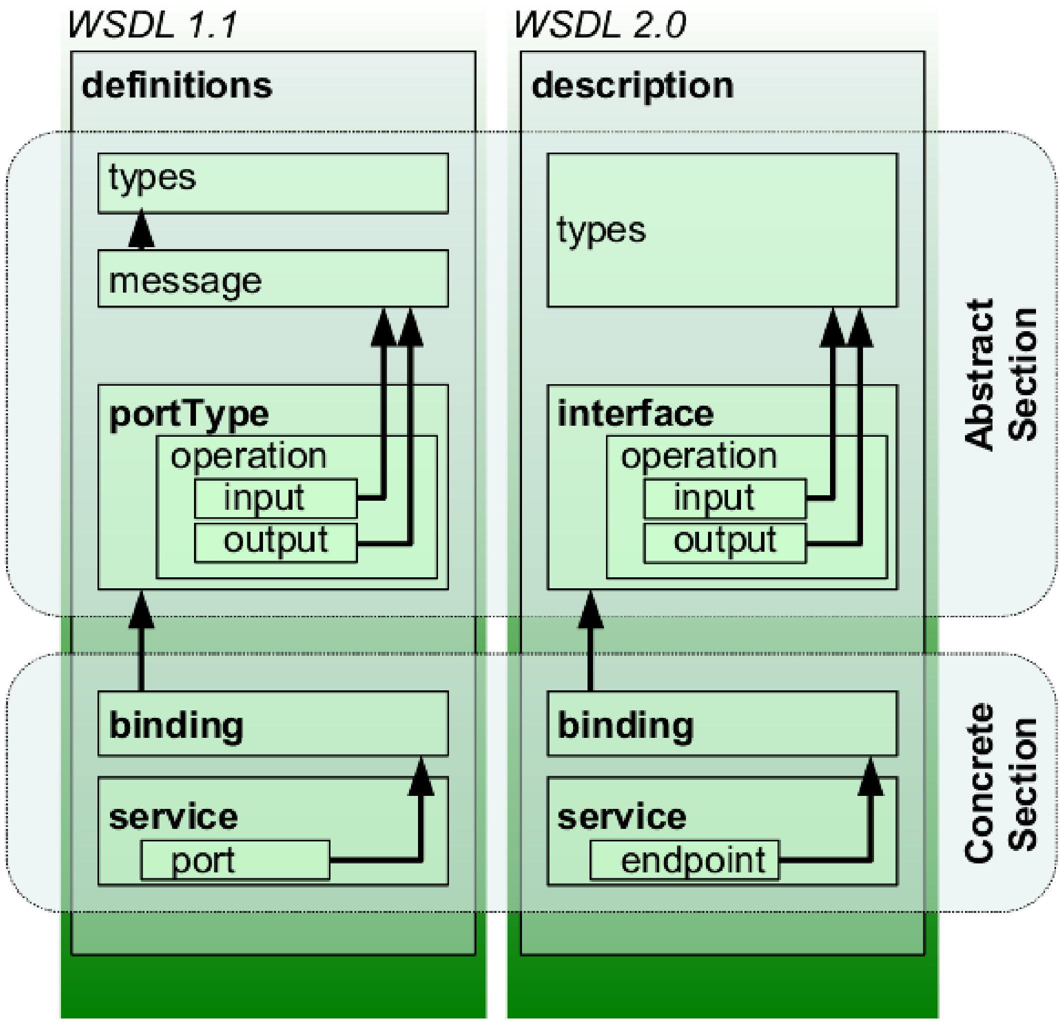
Other approaches with nearly the same functionality as RPC are Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM) or Sun Microsystems's Java/Remote Method Invocation (RMI).

Service-oriented architecture

Web services can also be used to implement an architecture according to service-oriented architecture (SOA) concepts, where the basic unit of communication is a message, rather than an operation. This is often referred to as "message-oriented" services.

SOA Web services are supported by most major software vendors and industry analysts. Unlike RPC Web services, loose coupling is more likely, because the focus is on the "contract" that WSDL provides, rather than the underlying implementation details.

Middleware analysts use enterprise service buses that combine message-oriented processing and Web services to create an event-driven SOA. One example of an open-source ESB is Mule, another one is Open ESB.



Representation of concepts defined by WSDL 1.1 and WSDL 2.0 documents.

Representational state transfer (REST)

REST attempts to describe architectures that use HTTP or similar protocols by constraining the interface to a set of well-known, standard operations (like GET, POST, PUT, DELETE for HTTP). Here, the focus is on interacting with stateful resources, rather than messages or operations.

An architecture based on REST (one that is 'RESTful') can use WSDL to describe SOAP messaging over HTTP, can be implemented as an abstraction purely on top of SOAP (e.g., WS-Transfer), or can be created without using SOAP at all.

WSDL version 2.0 offers support for binding to all the HTTP request methods (not only GET and POST as in version 1.1) so it enables a better implementation of RESTful Web services. However, support for this specification is still poor in software development kits, which often offer tools only for WSDL 1.1.

Automated design methodologies

Automated tools can aid in the creation of a Web service. For services using WSDL it is possible to either automatically generate WSDL for existing classes (a bottom-up strategy) or to generate a class skeleton given existing WSDL (a top-down strategy).

- A developer using a bottom up method writes implementing classes first (in some programming language), and then uses a WSDL generating tool to expose methods from these classes as a Web service . This is often the simpler approach.
- A developer using a top down method writes the WSDL document first and then uses a code generating tool to produce the class skeleton, to be completed as necessary. This way is generally considered more difficult but can produce cleaner designs

Criticisms

Critics of non-RESTful Web services often complain that they are too complex and based upon large software vendors or integrators, rather than typical open source implementations. There are open source implementations like Apache Axis and Apache CXF.

One key concern of the REST Web service developers is that the SOAP WS toolkits make it easy to define new interfaces for remote interaction, often relying on introspection to extract the WSDL, since a minor change on the server (even an upgrade of the SOAP stack) can result in different WSDL and a different service interface. The client-side classes that can be generated from WSDL and XSD descriptions of the service are often similarly tied to a particular version of the SOAP endpoint and can break, if the endpoint changes or the client-side SOAP stack is upgraded. Well-designed SOAP endpoints (with handwritten XSD and WSDL) do not suffer from this but there is still the problem that a custom interface for every service requires a custom client for every service.

There are also concerns about performance due to Web services' use of XML as a message format and SOAP/HTTP in enveloping and transport, such as that published by the University of Wollongong in 2005 by N.A.B.Gray.

Chapter-14

Server (Computing)

Several servers mounted on a rack, connected to a display

A rack-mountable server

In computing, the term **server** is used to refer to one of the following:

- a computer program running as a service, to serve the needs or requests of other programs (referred to in this context as "clients") which may or may not be running on the same computer.
- a physical computer dedicated to running one or more such services, to serve the needs of programs running on other computers on the same network.
- a software/hardware system (i.e. a software service running on a dedicated computer) such as a database server, file server, mail server, or print server.

In computer networking, a **server** is a program that operates as a socket listener. The term **server** is also often generalized to describe a host that is deployed to execute one or more such programs.

A **server computer** is a computer, or series of computers, that link other computers or electronic devices together. They often provide essential services across a network, either to private users inside a large organization or to public users via the internet. For example, when you enter a query in a search engine, the query is sent from your computer over the internet to the servers that store all the relevant web pages. The results are sent back by the server to your computer.

Many servers have dedicated functionality such as web servers, print servers, and database servers. **Enterprise servers** are servers that are used in a business context.

Usage

Servers provide essential services across a network, either to private users inside a large organization or to public users via the Internet. For example, when you enter a query in a search engine, the query is sent from your computer over the internet to the servers that store all the relevant web pages. The results are sent back by the server to your computer.

The term *server* is used quite broadly in information technology. Despite the many server-branded products available (such as server versions of hardware, software or operating systems), in theory any computerised process that shares a resource to one or more client processes is a server. To illustrate this, take the common example of file sharing. While the existence of files on a machine does not classify it as a server, the mechanism which shares these files to clients by the operating system is the server.

Similarly, consider a web server application (such as the multiplatform "Apache HTTP Server"). This web server software can be *run* on any capable computer. For example, while a laptop or personal computer is not typically known as a server, they can in these situations fulfill the role of one, and hence be labelled as one. It is in this case that the machine's purpose as a web server classifies it in general as a server.

In the hardware sense, the word *server* typically designates computer models intended for hosting software applications under the heavy demand of a network environment. In this client–server configuration one or more machines, either a computer or a computer appliance, share information with each other with one acting as a host for the other.

While nearly any personal computer is capable of acting as a network server, a dedicated server will contain features making it more suitable for production environments. These features may include a faster CPU, increased high-performance RAM, and typically more than one large hard drive. More obvious distinctions include marked redundancy in power supplies, network connections, and even the servers themselves.

Between the 1990s and 2000s an increase in the use of *dedicated hardware* saw the advent of self-contained **server appliances**. One well-known product is the Google Search Appliance, a unit that combines hardware and software in an out-of-the-box packaging. Simpler examples of such appliances include switches, routers, gateways, and print server, all of which are available in a near plug-and-play configuration.

Modern operating systems such as Microsoft Windows or Linux distributions rightfully seem to be designed with a client–server architecture in mind. These operating systems attempt to abstract hardware, allowing a wide variety of software to work with components of the computer. In a sense, the operating system can be seen as *serving* hardware to the software, which in all but low-level programming languages must interact using an API.

These operating systems may be able to run programs in the background called either services or daemons. Such programs may wait in a sleep state for their necessity to become apparent, such as the aforementioned *Apache HTTP Server* software. Since any software that provides services can be *called* a server, modern personal computers can be seen as a forest of servers and clients operating in parallel.

The Internet itself is also a forest of servers and clients. Merely requesting a web page from a few kilometers away involves satisfying a stack of protocols that involve many examples of hardware and software servers. The least of these are the routers, modems, domain name servers, and various other servers necessary to provide us the world wide web.

Server hardware

A server rack seen from the rear

Hardware requirements for servers vary, depending on the server application. Absolute CPU speed is not usually as critical to a server as it is to a desktop machine. Servers' duties to provide service to many users over a network lead to different requirements like fast network connections and high I/O throughput. Since servers are usually accessed over a network, they may run in headless mode without a monitor or input device. Processes that are not needed for the server's function are not used. Many servers do not have a graphical user interface (GUI) as it is unnecessary and consumes resources that could be allocated elsewhere. Similarly, audio and USB interfaces may be omitted.

Servers often run for long periods without interruption and availability must often be very high, making hardware reliability and durability extremely important. Although servers can be built from commodity computer parts, mission-critical enterprise servers are ideally very fault tolerant and use specialized hardware with low failure rates in order to maximize uptime, for even a short-term failure can cost more than purchasing and installing the system. For example, it may take only a few minutes of down time at a national stock exchange to justify the expense of entirely replacing the system with something more reliable. Servers may incorporate faster, higher-capacity hard drives, larger computer fans or water cooling to help remove heat, and uninterruptible power supplies that ensure the servers continue to function in the event of a power failure. These components offer higher performance and reliability at a correspondingly higher price. Hardware redundancy—installing more than one instance of modules such as power supplies and hard disks arranged so that if one fails another is automatically available—is widely used. ECC memory devices that detect and correct errors are used; non-ECC memory is more likely to cause data corruption.

To increase reliability, most of the servers use memory with error detection and correction, redundant disks, redundant power supplies and so on. Such components are also frequently hot swappable, allowing to replace them on the running server without shutting it down. To prevent overheating, servers often have more powerful fans. As servers are usually administered by qualified engineers, their operating systems are also more tuned for stability and performance than for user friendliness and ease of use, Linux taking noticeably larger percentage than for desktop computers.

As servers need stable power supply, good Internet access, increased security and are also noisy, it is usual to store them in dedicated server centers or special rooms. This requires to reduce power consumption as extra energy used generates more heat and the temperature in the room could exceed the acceptable limits. Normally server rooms are equipped with air conditioning devices. Server casings are usually flat and wide, adapted to store many devices next to each other in server rack. Unlike ordinary computers, servers usually can be configured, powered up and down or rebooted remotely, using out-of-band management.

Many servers take a long time for the hardware to start up and load the operating system. Servers often do extensive pre-boot memory testing and verification and startup of remote management services. The hard drive controllers then start up banks of drives sequentially, rather than all at once, so as not to overload the power supply with startup surges, and afterwards they initiate RAID system pre-checks for correct operation of redundancy. It is common for a machine to take several minutes to start up, but it may not need restarting for months or years.

Server operating systems

Server-oriented operating systems tend to have certain features in common that make them more suitable for the server environment, such as

- GUI not available or optional

- ability to reconfigure and update both hardware and software to some extent without restart,
- advanced backup facilities to permit regular and frequent online backups of critical data,
- transparent data transfer between different volumes or devices,
- flexible and advanced networking capabilities,
- automation capabilities such as daemons in UNIX and services in Windows, and
- tight system security, with advanced user, resource, data, and memory protection.

Server-oriented operating systems can, in many cases, interact with hardware sensors to detect conditions such as overheating, processor and disk failure, and consequently alert an operator or take remedial measures itself.

Because servers must supply a restricted range of services to perhaps many users while a desktop computer must carry out a wide range of functions required by its user, the requirements of an operating system for a server are different from those of a desktop machine. While it is possible for an operating system to make a machine both provide services and respond quickly to the requirements of a user, it is usual to use different operating systems on servers and desktop machines. Some operating systems are supplied in both server and desktop versions with similar user interface.

The desktop versions of the Windows and Mac OS X operating systems are deployed on a minority of servers, as are some proprietary mainframe operating systems, such as z/OS. The dominant operating systems among servers are UNIX-based or open source kernel distributions, such as Linux (the kernel).

The rise of the microprocessor-based server was facilitated by the development of Unix to run on the x86 microprocessor architecture. The Microsoft Windows family of operating systems also runs on x86 hardware, and since Windows NT have been available in versions suitable for server use.

While the role of server and desktop operating systems remains distinct, improvements in the reliability of both hardware and operating systems have blurred the distinction between the two classes. Today, many desktop and server operating systems share similar code bases, differing mostly in configuration. The shift towards web applications and middleware platforms has also lessened the demand for specialist application servers.

Servers on the Internet

Almost the entire structure of the Internet is based upon a client–server model. High-level root nameservers, DNS servers, and routers direct the traffic on the internet. There are millions of servers connected to the Internet, running continuously throughout the world.

- World Wide Web
- Domain Name System
- E-mail

- FTP file transfer
- Chat and instant messaging
- Voice communication
- Streaming audio and video
- Online gaming
- Database servers

Virtually every action taken by an ordinary Internet user requires one or more interactions with one or more servers.

There are also technologies that operate on an inter-server level. Other services do not use dedicated servers; for example peer-to-peer file sharing, some implementations of telephony (e.g. Skype), and supplying television programs to several users (e.g. Kontiki, SlingBox).

Energy consumption of servers

In 2010, servers were responsible for 2.5% of energy consumption in the United States. A further 2.5% of United States energy consumption was used by cooling systems required to cool the servers. It was estimated in 2010, that if trends continued, by 2020, servers would use more of the world's energy than air travel.