Table of Contents for Chapter 11

Part 2. Architect's Toolbox - 'bird's view of the terrain'

<<< … >>>

Chapter 11. More Platforms and Core Technologies

Information Technology implies that we must have a computer somewhere and some software running on it. This notion becomes so intuitive, that we do not think about miracles of IT infrastructure anymore.

Let's make a step back and take a look at underlying IT infrastructure that we take for granted.

We discuss types of hardware from mainframes or Big Iron, to PC and other client devices. In order to support a distributed application, these devices have to be inter-connected into computer networks.

Computer is driven by some kind of the System or Infrastructure Software – Operating System in first instance. Is not it a miracle how this invisible mythical software without flesh, smell and sound makes some nuts and bolts move?
Hardware and System Software stack provides a common computing platform on which we build Application Software that directly addresses the needs of the user. User, or the core business of the Enterprise, should not really concern themselves with the intricacies of hardware and software stack – but the Enterprise Architect should.

Chapter helps you sort out main types of the hardware, networks and system software there is. Some computer systems break the boundaries of common categories and defy attempts to classify them. Commonly accepted taxonomy in computing starts with notions of hardware and software. This distinction can get us into trouble right away, as some common software functionality may be implemented in hardware, and vice versa. But we have to start somewhere…

Hardware

Three general categories, types, or segments of computing hardware are *Servers*, *Server Appliances* and *Customer Premise Equipment* (CPE).
Specialized computing devices may fit into this segmentation easily. Alternatively, some particular multipurpose servers may be positioned to perform different functions in the Enterprise infrastructure, and cross the boundaries of defined segments.

Servers include back-end computing devices that possess high performance and capacity for number crunching and data processing. *Servers* may vary dramatically in performance and capacity, price, and computer organization.
Servers may belong (often arbitrarily) to *Supercomputers*, *Mainframes*, or *'Midrange' Servers* categories.

Supercomputers are high-performance computing devices designed for intensive computational operations or high–volume commercial functions. Supercomputers employ architectures with multiple CPU and parallel processing of computer operations. Supercomputers often used to perform specialized functions (like weather forecasts, aerodynamic models, geographical mapping, scientific computations etc.), and attract high costs of deploment.
Typical *Mainframe* computer (or, 'Big Iron') provides a platform for general-purpose Enterprise Information System, and serves many concurrent users.
'Midrange' Servers are realy a basket gategory for computers that arguably do not seem fit into *Mainframes* category. Indeed, high-end 'midrange' servers challenge conventional mainframes, and even supercomputers, on many levels. Innovations in hardware quickly enter the mainstream of computer organization in pursuit of competitive advantage. Tomorrow's 'midrange' servers routinely beat yesterday's mainframe benchmarks, and this process does not show any signs of slowing down.

Server Appliance is a networking and communication hardware device, performing usually specialized internetworking or access control function. Server Appliances may perform protocol and content filtering in a firewall, virus protection, caching of content, VPN and remote access management etc.

Customer Premise Equipment (CPE) include wide variety of client-facing computing devices – *Workstations*, *Personal Computers*, *Ultra-Portable* and *Embedded* computers and Personal Digital Assistants (PDA).

Workstations are typically represented by high-end CISC or RISC-based CPU architectures, with high-performance high-resolution graphics, integrated floating-point processing, integrated networking capabilities, and 32 or 64-bit multitasking OS.

Personal Computers (PC), their variations and descendants, penetrated the mass market, and reached the high level of acceptance and understanding in wider society. Personal Computers include desktops, as well as more compact (but not necessarily less powerful) notebooks or laptops promoting greater mobility of businesses, as well as of individual 'road warriors' of IT.

Miniaturisation of personal computing devices opens up further business opportunities by bringing the computing and communication power into the thick of the core business of the Enterprise, or into the infotainment endevour. *Ultra-Portable* and *Embedded* devices bring to the individual the personal computing capability where and when it is needed.
Ultra-Portable computing devices may include small lightweight laptops, handheld devices like Tablet PDA or Clamshell PDA, wearable computers etc. – sky is the limit (or, rather, limit is our imagination and the sense of purpose only).
Marriage of *Ultra-Portable* personal computers with mobile phones greatly increases the consumer qualities of personal computing by breaking the communication, performance, and service level barriers.
Embedded computing devices may bring unexpected intelligence to our home appliances, and new business, housekeeping, or entertainment possibilities.

Figure 11.1 depicts main types of computing hardware.

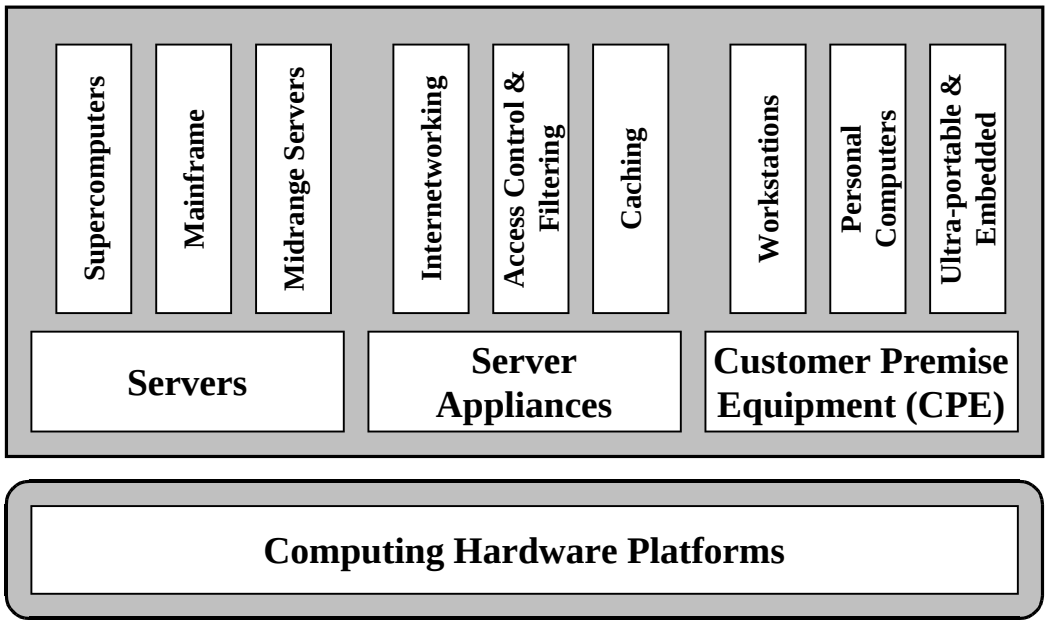

Figure 11.1. Computing Hardware Categories

Computer 'Bare Bones'

It always pays to remember how it all started in the history of computing hardware.

Very basic idea behind the computer was the automation of the execution of some *set of instructions* that perform data processing on some *set of data*, producing some valuable results in the form of data or control commands.
Furthermore, these *set of instructions* and *set of data* in the computer need to be easily replaceable or re-configurable, so that computing device able to do its job on other instructions and data as well – saving rebuilding the whole device for every new task.

This requirement of de-coupling the computer device from the *instructions* and *data* it operates on, lead to the concept of program and data stored in memory temporarily, only to be manipulated and replaced by other program and data when previous task is completed.
Alan Turing's paper "On Computable Numbers" in 1937 presented the concept of the Turing machine.
John von Neumann introduced the concept of a stored program in 1945.

Note that the abstract idea of the computer does not rely on any technology of its implementation.
First computers – calculators, Difference Engines, and Analytical Engines – were mechanical, and made of metal, wood, ivory, leather.
In twentieth century, computer technology swiftly progressed through to electro-mechanical devices, to the use of transistors and vacuum tubes, to ferrite-core memory and more compact electronic circuits, to circuitry encased in chips, to research in nano-technology and computing on molecular level.

Modern computers evolved from the initial fundamental computing ideas towards variety of complex and distributed architectures in response to the never-ending pressure to achieve better performance, capacity, reliability, parallelism, flexibility and fittness for the multitude of purposes, and commercial viability.
However, basic architecture of the single computer remains the same – stored programs and data, and some control device that will crunch numbers by executing stored commands one by one.

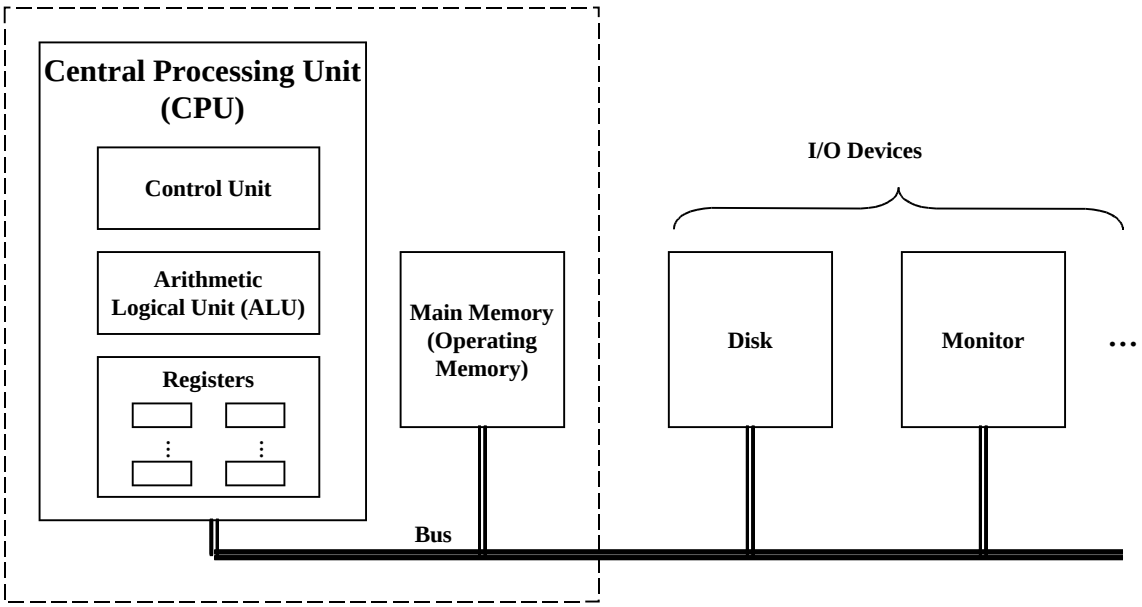

Figure 11.2. Basic Computer Architecture

Storage

Redundant Array of Inexpensive Disks (RAID)

RAID is a method of increasing the performance and fault tolerance of data stored on the multiple disk drives.
Multiple independent and inexpensive disk drives grouped together into 'array' to provide redundancy of the storage, and, at the same time, to present the storage to the processor as a single logical drive.

Initially, in early 1990's, RAID was implemented on high-end mainframes, but quickly made its way into broader market, including open systems and various mid-range platforms.

It is important to understand, that RAID is designed to improve fault tolerance against the storage failure within the array, and does not protect from the bigger disaster.

There are six 'official' RAID levels, and several hybrid implementations. Basic 'official' RAID levels are:

- ***RAID 0, also called "striping".*** Data blocks are split and written across multiple disks simultaneously. No redundancy or fault tolerance as such – just an increased I/O performance due to parallelism. Used where larger I/O bandwidth is required (like video or image processing), or in database applications with intensive random I/O. Requires N number of drives (ie. no redundancy).
- ***RAID 1, "disk mirroring".*** Provides enhanced read performance and 100% redundancy, with instantaneous recovery. Requires N*2 drives for full mirroring.
- ***RAID 2, parallel array with error correction code (ECC).*** Does not provide 100% duplication, limited benefits over RAID 3. Number of required drives varies, depending on implementation.
- ***RAID 3, simolar to RAID 2, but uses parity instead of ECC for fault tolerance.*** Uses parallel array with striping. One drive in array used to store parity data for recovery. Good for high data transfer, but is not optimised for parallel access in multiple user environment. Requires N+1 drive, with an aditional drive to hold parity data.

- ***RAID 4, like RAID 3, but with larger stripes to hold entire records.*** Supports only one write operation at a time. Inferior to RAID 5. Requires N+1 drives.
- ***RAID 5, "rotating parity array".*** Parity data is not stored on a separate dedicated disk, but distributed across array like the actual data. Faster write performance due to overlap of stripes and parity data. Suiteble for database and online transaction processing applications. Requires N+1 drive, equivalent of one extra drive (not dedicated, though) for the parity data.
- ***RAID 6, with dual redundancy.*** Allows for two lots of parity data that are computed differently. Increased fault tolerance with some additional overhead. Recovery of lost data is possible even if two drives fail at the same time. Requires N+2 drives, equivalent of two extra drives, for two levels of parity data.

Data protection in RAID can be implemented either in hardware, or software.

Hardware-based RAID is more common, and uses dedicated storage processors.
Software-based RAID passes the overhead on to CPU. Because software-based RAID cannot protect the data if system is not booted, protecting the drive that stores the operating systems may become an issue.

Storage Area Network (SAN)

Available options for storing our precious digital assets are becoming increasingly comprehensive and flexible, and evolve rapidly away from the simple notion of the "hard drive" that is directly attached to the computer.
Storage has become a self-sufficient and complex infrastructure of its own, and provides a shared repository of data for the servers on different platforms such as various incarnations of Unix, flavours of Microsoft Windows, IBM AS/400, NetWare.

In the context of more complex storage configurations discussed here, familiar "hard drive" connected by a cable to a processor is called now a Direct Attached Storage (DAS).
If storage device appliance consisting of disc storage and own processor is connected to the LAN for other servers to share this resource, we call it Network Attached Storage (NAS). NAS device is a stand-alone device attached to network, but does not have a dedicated network. NAS provides ease of storage management.

Storage Area Network (SAN), as the name implies, resides on a dedicated network that hosts a pool of shared storage devices (see Figure 11.3). SAN configuration provides improved overall performance, scalability, control, and backup facilities.

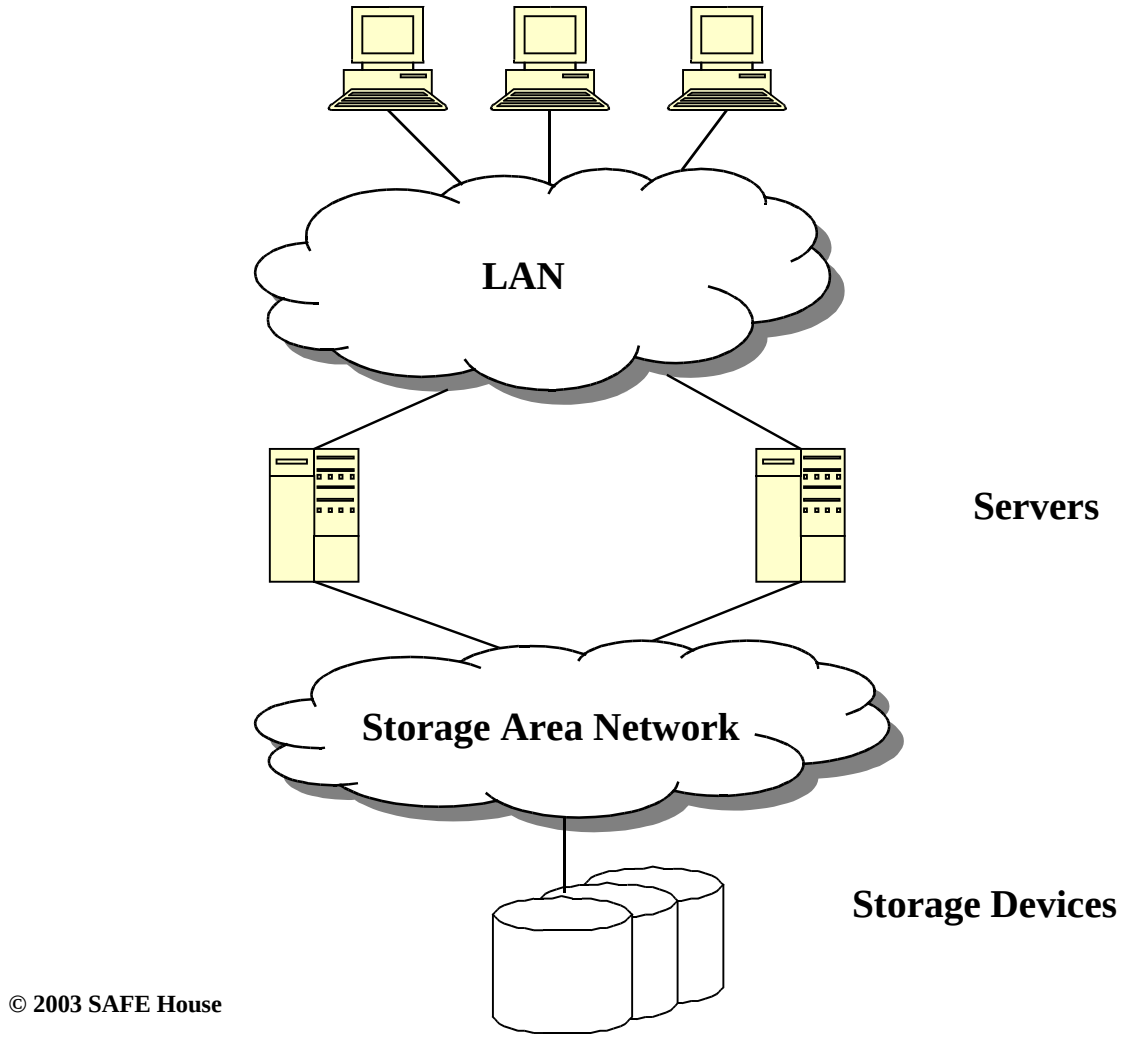

Figure 11.3. Storage Area Network

SAN devices may use Fibre Channel media for any-to-any connections between processors and storage on the SAN network. However, Ethernet media using iSCSI protocol is emerging.

In addition to DAS, NAS and SAN network connectivity schemas, to understand available storage options better, we need to understand the types of *media* (physical wiring and associated low level protocol) and *I/O protocol* (how I/O requests are executed over the media).

Some standard types of *media* for connecting storage to processor are:

- ***Ethernet***. Ethernet is a media and its protocol, initially introduced for building LANs in the 1980s. IP-based protocols such as TCP/IP generally run on top of Ethernet. Typical bandwidth up to 1Gbps
- ***Fibre Channel***. Fibre Channel is a technology developed in the 1990s. Bandwidth up to hundreds MBps is expected
- ***Parallel SCSI (Small Computer Systems Interface, pronounced "scuzzy")***. SCSI is a media and a protocol, suitable for the short-range connections (25 metres or less). Bandwidth generally reaches up to 160MBps
- ***Serial Storage Architecture (SSA)***. SSA is a media for connecting disks in some disk systems. Bandwidth is 160MBps

Types of *I/O Protocols*:

- ***SCSI***. Block-level I/O Protocol for issuing I/O Commands to disk. SCSI I/O Protocol can run over different types of media (not just over Parallel SCSI). ***iSCSI*** protocol runs SCSI block I/O Commands over the TCP/IP network

- ***NFS (Network File System)***. File-level, device-independent protocol, originated in the Unix world
- ***CIFS (Common Internet File System, pronounced "siffs")***. File-level, device-independent protocol, originated in the Microsoft Windows NT world

SAN configuration has distinct benefits for the Enterprise Architectures including remote access to the storage by multiple processors, consolidation of the storage into shared and scaleable pools of disks and tapes, improved manageability of massive shared storage, and offloading the I/O traffic from a LAN to a dedicated network.

Communications and Networking

Bridges, Routers and Gateways

Networks and communication equipment may belong to the region or the enterprise. One part of the enterprise wants to connect to other. One enterprise wants to get an access to some global network in order to establish a communication with another enterprise.
In other words, networks need to connect and talk to other networks.

Bridges, *Routers* and *Gateways* allow network connectivity - remote network access between geographically distributed networks and interconnections between separate networks, network partitions, segments and sub-networks.

Distinction between *Bridges*, *Routers* and *Gateways* is better understood in relation to the OSI and DARPA reference models, and TCP/IP protocol stack that introduce networking layers (see Figure 8.2).

Bridge connects LAN segments at the Network Interface Layer level (DARPA) or Physical and Data Link levels (OSI).
Bridge is independent of any higher-level protocol and, as such, is transparent to IP.

Router interconnects networks at Internetwork Layer level (DARPA) or Network Layer level (OSI) and routes packets between networks. *Router* attached to two or more physical networks and forwards datagrams between the networks.
Router implements the IP Layer of TCP/IP protocol stack, and more sophisticated than IP packed routing functions. *Router* understands the IP addressing structure and makes decisions on forwarding packets. *Router* is able to select the best transmission path and optimal packet size.
Router is said to be visible to IP. Because of this, the term *IP Router* is often used. Some confusion may be caused by older terms for *Router – IP Gateway*, *Internet Gateway* and *Gateway*. However, term *Gateway* is normally used for network connections at a higher levels.

Gateway interconnects networks at higher layers than *Bridges* and *Routers*. *Gateway* is said to be opaque to IP. That is, a host cannot send an IP datagram *through* a *Gateway* – it can only send it *to* a *Gateway*. The higher-level protocol payload carried by the datagrams is then passed on by the Gateway using whatever networking architecture is used on the other side of the *Gateway*.
Gateway may provide transformation of the data and limit the interconnectivity between two networks to a subset of communication protocols.

Concept of *Firewalls* is closely related to *Routers* and *Gateways*, with the additional spin of adding security and access control to the network traffic from and to the untrusted network or Internet.

Software

Software components, and ways they interoperate, are the main building blocks that Software Architect use in constructing the Enterprise Architecture to satisfy business needs.
Intimate understanding of software categories, technologies, and products provide a basis for the day-to-day contribution of the Software Architect. Knowledge of software landscape is a starting point in building the Enterprise Architecture, a tool of the Software Architect.

Like in any profession, Software Architect may start learning how to use the tool, when he or she has one.

Figure 11.4 presents main categories of software in the overall software stack, from System Software controlling the resources of the hardware platform, to variety of Application Software that directly addresses the needs of users.
We can use computers for variety of purposes, and figure cannot possibly capture all of them. Category *Application Software* provides a generic basket for all types of applications that we neglected to mention here so far.

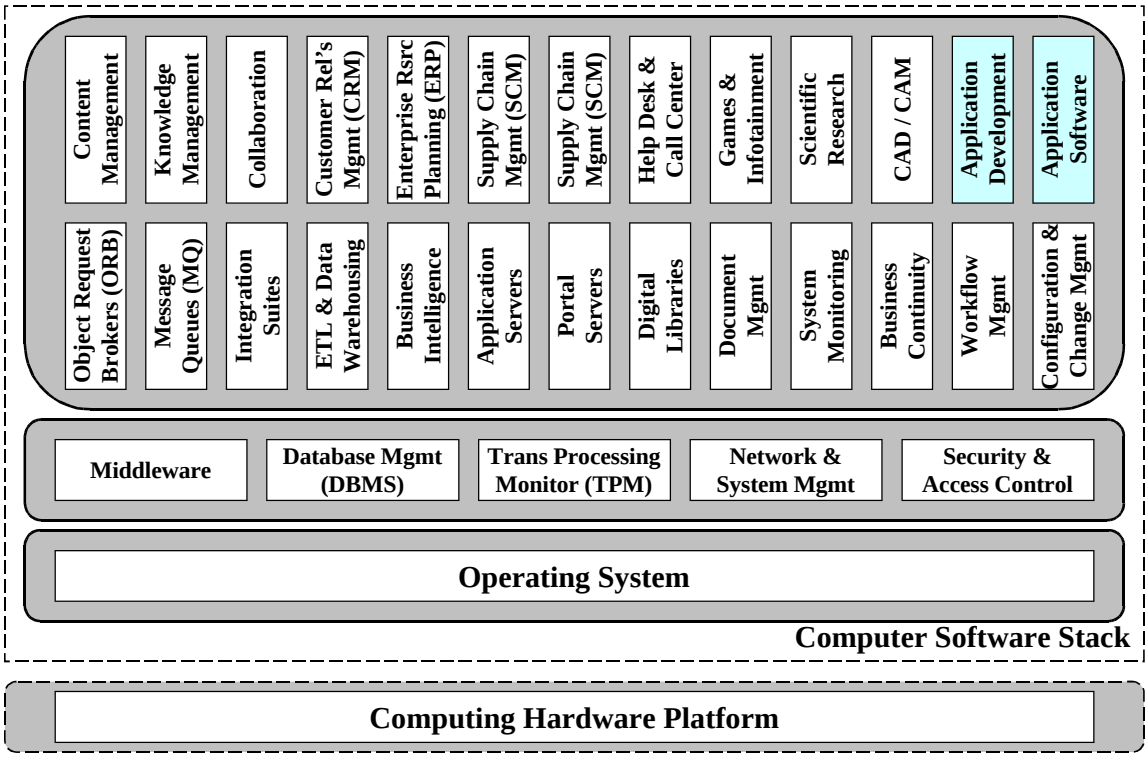

Figure 11.4. Computer Software Categories

Operating System (OS) – touching the bare hardware, or, where miracle happens…

Notion of stored executable program gave birth to the very idea of *software*, computer programming, and, eventually, Software Architecture.

There is very long way from the 'bare bones' computer *hardware*, through some kind of *system software*, to the *application software* that fullfills specific business needs of the user.
Software presents a stack of layers and components that delivers raw number-crunching capability of the hardware all the way to the business user of the computer power and software functionality.

On the lowest level of the software stack, where software meets the hardware (or, if you like, rubber hits the road), is the layer of system software that controls the resources of the computer and execution of programs, called *Operating System*.

As we shall see, there are other kinds of system software, like Database Management Systems, Transaction Processing Monitors, Application Servers etc. (somewhat more *application* and less *system* pieces in a software stack). However, it is *Operating System* that brings the computer hardware to life and makes it do some useful work for other software, and, eventually, for us.

Some Operating Systems may be able to run on different types of computer architectures – flavours of Unix and Linux, for instance.
Other Operating Systems are tightly coupled to the specific hardware architectures - Windows, z/OS, OS/400, Mac OS.

Dominant mainframe OS is IBM's z/OS, descendant of MVS and OS/390.
Midrange servers mostly run on flavours of Unix and Windows.
Personal computers run on Windows (IBM-compatible), Mac OS (Apple PCs), or Linux.

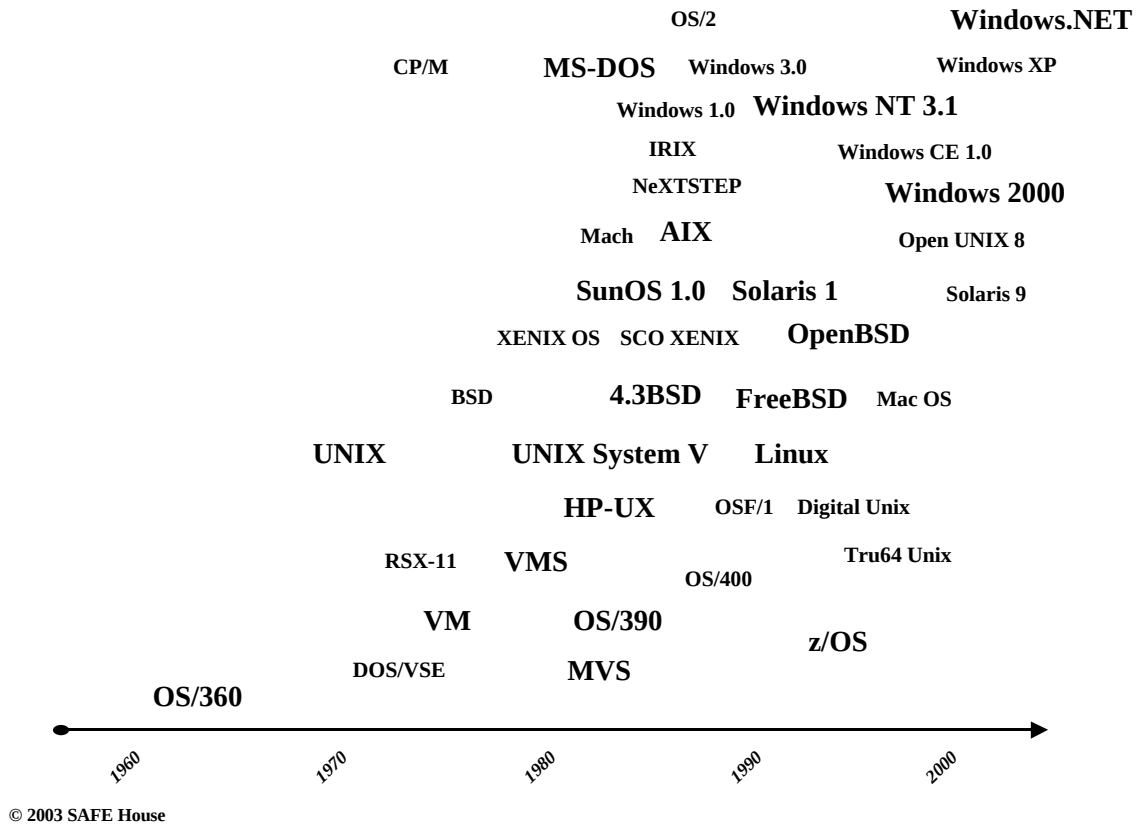

Figure 11.5. History of Operating Systems

Programming Languages – from source code to execution

Programming Language is the language that humans made up for themselves to describe business processes, and, at the same time, to be able to convert these process descriptions for execution on computing device, into form understandable by computers.

Thus, *Programming Language* tries do a balancing act, and to serve well to two Gods at once – Human and Computer. Inherently, these two have a vastly different view of the world. *Programming Language* bridges the semantic and syntax gap between two visions.

Needless to say, the more carefree and sloppy we want to be in articulating our wishes to the computer, the more sophisticated and clever this computer should be, and more complex the process of translation into native computer language becomes.

It is a never-ending mistery that fascinates and motivates generations of Computer Programmers - ability to deduce in your mind some abstract process, capture it in the *Programming Language*, and to make computer perform some work following your instructions, with very tangible material results. You feel as if computer becomes an extension of your body, like another limb.

Programming Languages evolved dramatically since first storable in memory sets of execution instructions in early computers.
Then and now, 'bare bones' computer hardware natively understands only language of 1s and 0s. At least one participant in the dialog – Human – cannot possibly be too happy about this language of communication.

First programmers actually wrote programs in binary 1s and 0s.
As programmer's folklore says: laziness motivates innovation - so that programmer could get away with doing less work for the same outcome. And very first programming innovation was the hexadecimal presentation of the binary program code, wnen one hexadecimal number (0 to F) could represent four binary 1s and 0s at once – imagine savings in programming efforts!
Hexadecimal writteup (and, later, printout) of the memory contents was called a memory dump.

Next language improvement was allocating the letter mnemonic to the hexadecimal codes of computer instructions.
Mnemonic resembled abbreviated meaningful words that denoted what computer operation actually supposed to do, and was much easier to remember.
Whole computer instruction consisted of operation name, and operation's expected arguments – data parameters, memory addresses, or mnemonic labels for data or memory addresses, like:

```
Op_Name     <Arg1_Address>, <Arg2_Address>
```

For instance, Operation Name *Op_Name* could be *ADD* for *addition*, and instruction would require adding argument *Arg1* found in memory address *Arg1_Address* to argument *Arg2*, and saving result in memory address *Arg2_Address* (thus overriding old value of the argument *Arg2*).

This basic mnemonic representation of the lowest-level binary machine language called Assembler. Every architecture of computer organization has its own Assemblere, tied to the set of machine operating instructions.

Despite being a noticeable improvement over hexadecimal numeric program code, Assembler still had a long way to go towards the natural human language. Our lazy Programmer again had something to think about.
Next stage was to encode computer instructions into syntax that better resembled the human language.

In general, Programmer writes the *Source Code* of the program in a chosen *Programming Language*. Source Code is just a text file, readable by human and, more importantly, understood by human Programmer who is familiar with the *Programming Language*.
Source Code of the program contains instructions for the computer to perform some business transaction.

Source Code of the program has to undergo some translation in order to convert the program into the set of basic low-level machine instructions understood by computer.

In general, this process of translation of the program Source Code into set of machine instructions can take one of two possible paths:

- Program is translated into machine code and stored for execution at some time later. This process is called *compilation*, and system program that does it to the Source Code called *Compiler*. *Compilation* is separated from the execution of the program, and *Compiler* can afford to take its time, possibly make severall passes over the program code, and optimise the resulting machine code better

- Program is translated into machine code and executed ‘on the fly’. System program that reads the Source Code of your program and starts executing the program immediately while reading it, called *Interpreter*. You do not have to store the translated program – just pass the Source Code of the program to the *Interpreter* again next time when you wish to execute the program. Translation time becomes part of the program execution and contributes to runtime latency. Most scripting languages follow this scenario

Figure 11.6 depicts the first scenario – general process of program development using *Compiler*.
This process may vary in details from one *Compiler* to another, and vary in different Operating Systems.
For instance, some *Compilers* may produce intermediate assembler code that needs to be compiled further into machine code by the *Assembler* program.

Single program likely will require many standard programs to complete the whole work (like timestamp formatting, or data retrieval – you won’t write this code every time yourself).
Operating Sytem may fetch these modules and bundle them with your program before actual execution (‘early binding’, see figure), or fetch these standard modules at runtime (‘late binding’).
Late binding of the standard modules from the shared executable libraries at runtime improves manageability of application software, and saves a lot of storage for the executable programs – system standard modules stored only once in one place.

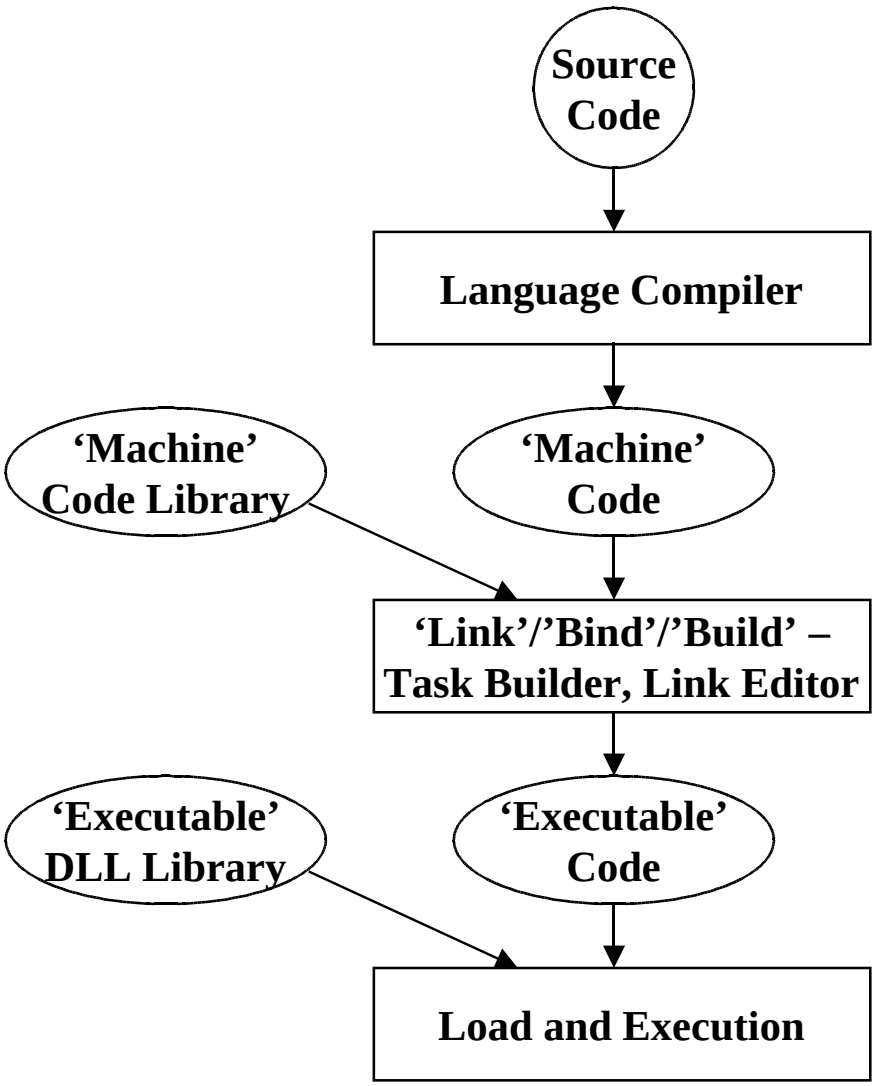

Figure 11.6. Program from Source Code to Executable

Program preparation life cycle looks quite simple, if only programmers never made mistakes.
This cycle, or some of its steps, will be repeated many times during program 'de-bugging'. Many programmers visualise hunted program errors as bugs somewhere in the source code.
However, in the history of programming, term *bug* for the program error originated from the real moth that short-wired the circuit in the early computer, and caused a lot of grief to the programmer [WWW Hopper]. Grace Hopper kept this actual moth – the very first program *bug* – in her diary.

Hundreds of programming languages were invented, for various purposes, and with different fate. *Programming Languages* evolved with our growing understanding of what makes the software good – from structured and modular programming to object-oriented programming.

Figure 11.7 shows timeline with history of more prominent *Programming Languages*.

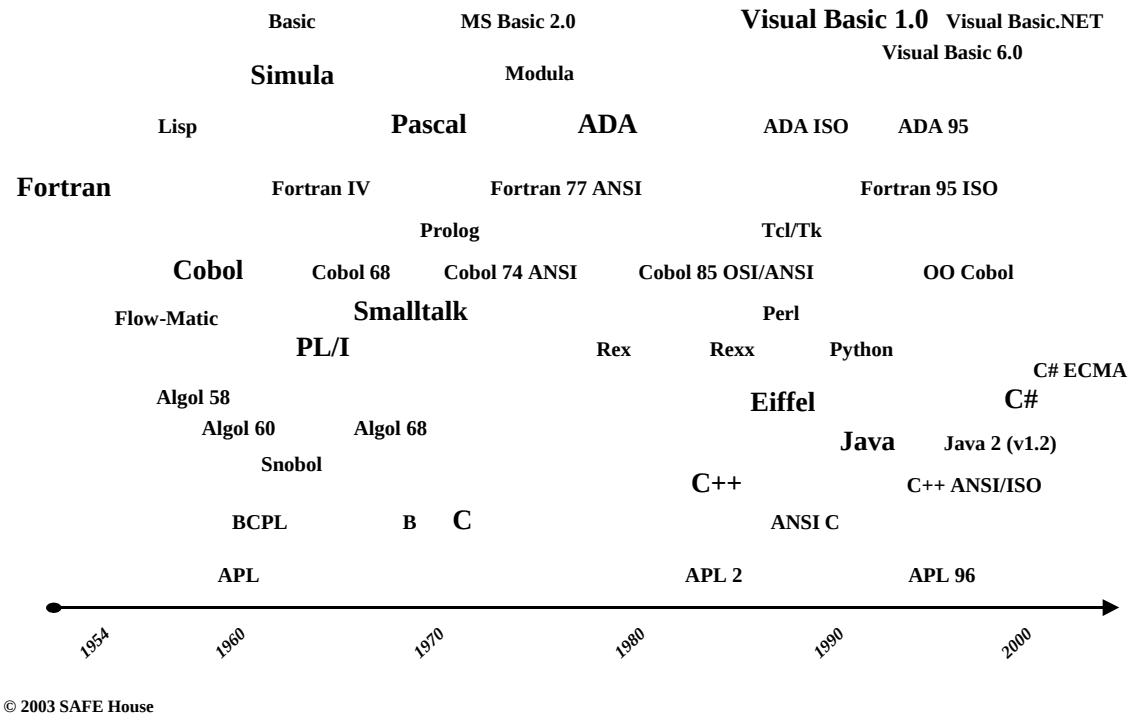

Figure 11.7. History of Programming Languages

Database Management Systems (DBMS)

Database Management System is the software product that is used primarily for the storage and retrieval of structured data.
We say *primarily*, because modern databases exhibit capabilities for transactional processing, security, data transformation, and managing complex business logic (using stored procedures and database scripting). It is becoming increasingly difficult to categorize overlapping components of the *middleware*.

In general, database architectures implement a commonly accepted ANSI layered view on data structures with distinct *physical* data model or schema (with mapping of data structures onto physical storage and access methods), *logical* or *conceptual* data model or schema, and customer *views* of the database logical model.
Depending on the pre-dominant type of data structure of the logical data model, databases may be described as relational, hierarchical, network, or object databases. Existing products may implement a mixture of features. For instance, relational databases may exibit some capabilities of object databases, or facilitate the object-relational mapping.

Relational databases captured the lion share of market and mindshare of developers.
DBMS market has undergone a period of consolidation and acquisitions. Major remaining products on the Relational DBMS market are IBM's DB2, Microsoft's SQL Server, and Oracle's Oracle.
Informix (acquired by IBM), Ingress (acquired by Compurter Associates), and Sybase remain notable relational database players, but only just.
Freeware databases, like MySQL and PostgreSQL, provide a viable alternative for cost-conscious players. These databases proved to be very popular with ISPs, small developers, with more agile and less mission-critical applications.

Relational databases use SQL as a common language for data definition and data manipulation.

Pre-relational DBMSs (like IBM's IMS), and non-relational (mostly Object Databases) do not enjoy significant volumes of new deployments.

Transaction Processing Monitors (TPM)

Transaction Processing Monitors (TPM), as name implies, perform resource management of shared computing resources, and provide a transaction-safe (see *ACID*) and secure runtime environment for simultaneous execution of many processes, tasks, or components.
This conventional definition of TPM is being challenged by evolving capabilities of products on the market, and needs a better qualification fot the specific TPM market segment or product.
Really, Operating Systems, Database Management Systems, Application Servers, Request Brokers and Message Buses, and Workflow Engines routinely perform transactional resource management function, stealing thunder from the dedicated Transaction Processing Monitors.
TPM capabilities are being absorbed by other system software components.

However, there are high-end dedicated TPM products on the market, and we briefly describe them in this section.

IBM is prominently represented on the TPM market by flagship CICS, as well as IMS/DC (TPM*ish* component of the IMS database), and Encina (originally from Transarc).
CICS was initially developed in 1960s for mainframes. CICS experienced constant improvements and enhancements ever since the initial release, and reached enviable level of maturity, dependability, and dominance in high-end TPM market. In addition to the IBM's 'big iron' (or mainframes), CICS is available on other platforms, including AS400, AIX and Unix, OS/2, and Windows.
Encina was developed in 1980s by the research team at Carnegie Mellon. Encina is based on DCE concepts. IBM purchased Encina in 1994.

BEA ships TUXEDO and TOP END, originally Unix-based, but currently ported to dozen of other platforms. TUXEDO and TOP END TPMs combined still lead in Unix market share.
TUXEDO was originally designed as Unix-based replacement for IMS/DC, and was released in early 1980s. TUXEDO served as basis for several Open Group (X/Open) standards.
TOP END was released in early 1990s at NCR.

Tandem ships Pathway/TS fault-tolerant TPM, released in 1980s. Pathway/TS has 'personalities' for CICS and TUXEDO.

ACMS from Compaq (originally from Digital) was released in 1980s for VMS. Currently, ACMS has Unix and Windows versions available.

Microsoft's MTS is based on DCOM, and integrated with IIS Web Server and MSMQ Message Queue. MTS runs on Windows platform only.
MTS does not support persistent objects, and strongly encourages stateless servers. As far as TPM concerned, stateless servers significantly simplify its life and improve scalability, provided you are happy with handling persistent state otside TPM jurisdiction (eg. in database, file system, message queue, or external cache).

Transaction Processing Council (TPC) [WWW TPC] develops industry bencmarks for the evaluation of TPM performance – TPC-A, B, C, D, W, …
Due to the architectural differences in TPMs, performance benchmarks should recognise application requirements end-to-end, including handling of caching and persistence in or outside TPM. Otherwise, we may end up comparing apples and oranges.
For instance, follow the debacle about benchmarking and comparison of the demo Pet Store application implemented in J2EE (various implementations), and in Windows. E.g., see [WWW TSS].

Application Servers

Attempt to define the *Application Server* presents a surprisingly difficult challenge.

Application Server is an intrinsic part of *Middleware* in its broadest sense (see section on *Middleware*), and implements, or integrates with, common features and services that we come to expect from *Middleware* in the enterprise.

Commonly accepted expectations of what we get from *Application Server* include such functions as:

- Container for multitute of executing Components at runtime
- Thread and Connection Pooling for many concurrent processes to many resources
- Resources allocation and management, acquisition and release of Locks on resource handles
- Memory Management, garbage collection, page swaps
- Session and State Management
- Load Balancing, Failover, Caching
- Transaction Management for ACIDity of business transactions
- Security, Access Control and Personalization
- Adaptors, Interfaces and Connectors to external resource managers and application frameworks – DBMS, TPM, Message Queues and Object Request Brokers, Enterprise Suites and Portals. (Note that Application Servers may absorb some or all of these functions, possibly going far beyond their initial purpose)
- Presentation capability for rendering and delivery of results to users and client processes, including HTML page generation, HTTP request/response, XML/XSL data serialization
- Ability to develop more components implementing some new business logic. Development tools, re-useable components and libraries may be tightly coupled with Application Server

Armed with this definition of the *Application Server*, we can point to the most representative examples – J2EE-compliant Application Servers, and Microsoft's DCOM/MTS platform.

J2EE-compliant Application Servers, with multitude of implementations from variety of vendors, exhibit fierce individuality wherever possible outside and beyond the certified boundaries (often ahead of standard processes for competitive advantage, in anticipation of pending standard being eventually ratified).
In reality, J2EE Application Servers turning into vendor's Enterprise Integration Suites, and include components, portal solutions, adaptors, integration frameworks, and development tools that may be specific to this vendor. Strict J2EE compliance of your deployed application suite may become a balancing act, and will require special efforts from the Enterprise Architect.

Middleware: Buses, Component Brokers and Message Queues

In the broadest sense, *middleware* is any software, hardware, or network communication component that facilitates control and data flow between server and client in the transaction.
With geographically distributed servers and clients, middleware components may collaborate in variety of ways, and be physically partitioned and distributed in many different deployment patterns.
One may say - challenge of the Enterprise Integration Architect is all about *middleware*.

Specific discussions of *middleware* usually call for better qualification and narrowing down to the particular middleware component under consideration.

Middleware provides the integration glue that enables communication between servers and clients in conducting the business transaction between them. Participants in transaction are possibly distributed and remote, and possibly heterogeneous (i.e. implemented on different platforms and/or adhering to different interfacing standards and protocols, thus introducing an *impedance mismatch* between communicating participants).
Middleware shall bring client and server of the transaction together by establishing the communication access path and by determining the mutually agreed and supported protocol for the conversation.

Message Buses and Object Request Brokers (or, Component Brokers) provide the networking and interfacing infrastructure for establishing the conversation between components in the distributed computer system.
Examples of commercial message buses – various implementations of OMG CORBA standard, Microsoft's DCOM, Tibco's Rendezvous, SeeBeyond.

Prominent representative of CORBA implememtation is Iona's Orbix and E2A ('End to Anywhere') platform.

Figure 11.8 depicts the general concept of the Message Bus.
Note Message Bus plays the role of the conduit or message carrier between distributed components. Message Bus establishes conversation (when messages are being passed back and forth), or "request/reply" two-way interactive communication, or simple one-way message transfer.
In these scenarios, Message Bus relies on components in client and server endpoints of transaction to be up and running. Message Bus or Object Request Broker only responsible for delivery of the message, and does not store transmited messages in-transit for delivery later, in case server was not available. I.e., Message Bus supports the *synchronous* communication.

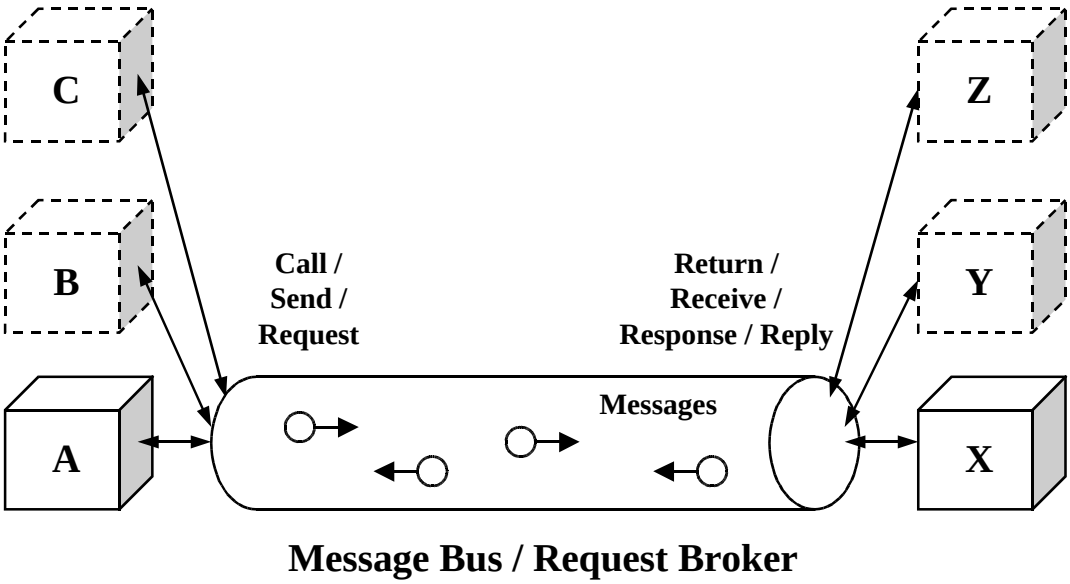

Figure 11.8. Message Bus and Object Request Broker Models

Unlike Message Bus or Object Request Broker, Message Queue stores messages from the supplier process or component, and delivers stored messages to the consumer sometime later.
Message can be deferred, or delivered almost instantaneously, possibly creating an illusion of synchronous communication. Still, communication using Message Queues is *asynchronous* by nature – Message Queue can only fake synchronicity.

Message Queues may be transient (stored in memory), or persistent (stored on disk).
Transient Message Queues will ensure better performance and latency, but may irrevokably lose valuable messages in case of failure, and call for more careful capacity planning.
Persistent Message Queues will ensure better recovery, at expense of greater performance challenges.

Prominent Message Queue products on the market are IBM's MQSeries, Microsoft's MSMQ, SonicMQ from Sonic Software. In Java world, JMS interface provides common wrapper to various implementations of Message Queues.

Figure 11.9 depicts Message Queue Communication Models. There are many possible scenarios how you can leverage Message Queue in your application.

Architect may decide to build application around *push* or *pull* model in Message Queue, or *publish-subscribe*, or *collect*, or *broadcast*, or the combination of all of the above.
Message Queue may *collect* and *publish* messages from one or many suppliers in the same queue. Also, one or many consumers of the messages may *subscribe* or receive *broadcast* from the same queue.

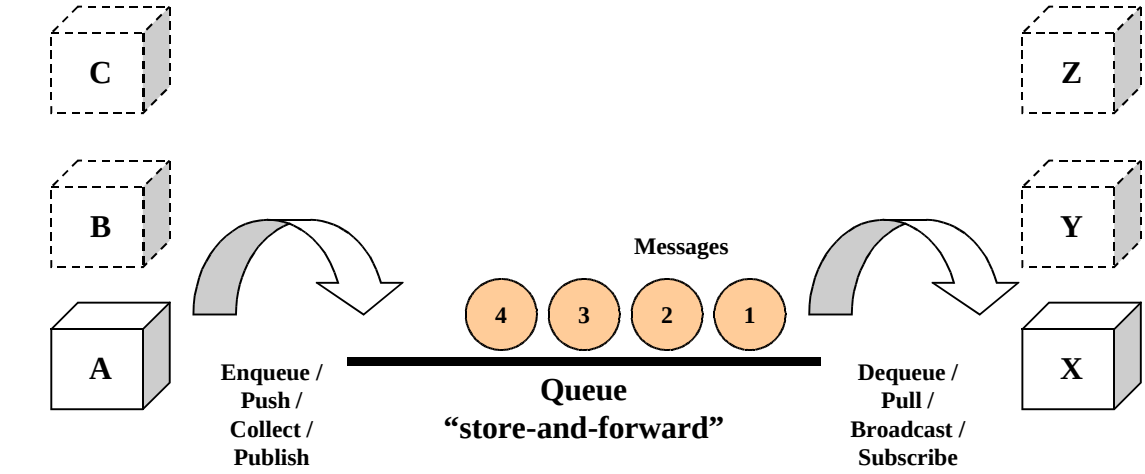

Figure 11.9. Message Queue Communication Models

Open Source Movement

In general, vibrant IT Developer's community proudly exhibits creativity, talent, and flair. Skillful programming requires significant and ever growing knowledge, as well as special abilities for unorthodox thinking and problem solving.
Software engineering cultivates the special type of temperament too – meticulous and thorough, yet passionate and excitable.

Commercial shrink-wrapped software, with its license fees and sometimes-questionable value or quality, may be perceived as stifling the free spirit and creativity of software developers.

Open Source movement was conceived as a natural outlet for software developers to vent their frustrations, and to provide a viable alternative to the monopoly of commercial offerings. Professional satisfaction, respect of peers and the awe of admirers played its part too.
In the *Open Source* model, innovation and collaboration flourish, probing and resourceful minds quickly find ways to achieve stated goals, to deliver sought-after functionality. Resulting software is given away for free, often together with the source code.
In the past, common perceived downside of the *Open Software* was its patchy support and unpredictable quality. However, word of mouth (keen users are the best testers), rapid fixes and improvements, and readily available help from the knowledgeable and interested community, all contributed to the agility and trustworthiness of the *Open Software*.

There are some common variations of the *Open Software* model:

- ***Freeware*** – software is given away at no cost. Source code may or may not be included.
- ***Shareware*** – software is given away on try-before-you-buy basis. Some fee is expected after the trial period. The trial version of software may be incomplete ("crippleware"), may contain

the expiry date that disables the software after the trial period finished, or the full version of the software (possibly with annoying alerts, “nagware”).

- ***Public Domain*** – the developer relinguishes all rights to the software. Anyone can modify the product and even resell it.
- ***Open Source*** – freeware with the source code.
- ***Copyleft*** – the antithesis of the copyrighted source code. Source code is in the public domain and can be modified, but cannot be sold.

Many software products in every domain and on every platform follow the Open Source model, from the Operating System to highly specialized utility program.
Many developers achieved fame and glory by their contributions into the Open Source. Just to mention a few names – Linus Torvalds (created Linux operating system in 1991, while he was a student in Finland), Richard Stallman (creator of GPL – General Public License, and founder of Free Software Foundation), Eric Raymond (founder of Open Source Initiative), and Bruce Perens (developer of Debian free software guidelines).

Open Source software is entering the mainstream IT industry, making a significant impact on the commercial products it is positioned against.
In some instances, when commercial offerings cannot find success of their own, these products may be released on the market as an Open Source, or bundled with other products “for free”. Alternatively, popular Open Source software may re-appear as a commercial product that may attract license fee, support fee, professional services fee, or otherwise extract money from the user for the privilege (eg., by charging for the full “enterprise” version of the software).

Clearly, Open Source software is not for everyone. Enterprise should understand all risks and, possibly hidden, costs associated with the Open Source.
With growing maturity and adoption of the open standards, Open Source solutions should not be discounted by the enterprises outright. And with big guns lining up behind the Open Source projects, viability of the Open Source for the enterprise improves. For instance, Linux enjoys a good momentum in development and support by large vendors like IBM.

In summary, decision to commit enterprise to the Open Source solution is the balancing act between all costs and risks associated with opposing commercial and Open Source products. Budget pressures, combined with growing maturity and acceptance of the Open Source software, may well tip the scales towards the viable Open Source product.

<<< … >>>