Definition > a distributed system is a collection of independent computers that appears to its users as a single coherent system

> hardware: machines are autonomous

> software: users think they're dealing with a single system

Characteristics:

> Differences between various computers and the way they're connected are hidden from users

> easy to expand/scale

> normally is continuosly available

Open distributed system -> system that offers services according to standard rules that describe the syntax and semantics of those services.

Computer networks [standard rules] -> formalized in protocols

Distributed systems [services] -> specified through interfaces (Interface Definition Language)

Interface definitions written in IDL nearly always capture only the syntax of services. [names of the functions, types of the parameters, return values, possible exceptions. These allow:

- an arbitrary process that needs a certain interface to communicate to another process that provides that interface.

- Two independent parties to build completely different implementations of those interfaces, leading to two separated DS that operate in the exact same way.

Proper specifications:

-> Complete: everything that is necessary to make an implementation has, indeed, been specified

-> Neutral: specification don't prescribe what an implementation should look like

Are important for:

Interoperability: two implementations of systems from different sources can co-exist and work together since both rely on each other's services pre-specified by a common standard.

Portability: an application developed for a DS A can be executed, without modification, on a DS B that implements the same interfaces as A

Another goals:

>Flexible: easy to configure the system out of different components possibly from different developers

>Extensible: easy to add new components or replace existing ones without affecting those components that stay in place

> Users and apps shouldn't be able to say when parts are being added, replaced or fixed

In order to support heterogeneous computers and networks while offering a single system view:

> DS are organized with a layer of software that's logically placed between a higher-level layer consisting of users and applications

> A layer underneath consisting of operative systems

This kind of distributed system are called Middleware.

If the system looks and acts like a classical single-processor timesharing system, it qualifies as a distributed system.

A distributed system should:

- Easily connect users to resources

- Hide the fact that resources are distributed

- be open

- Be scalable

If a DS is able to present itself to users and apps as if it were only a single computer -> the DS is transparent

Types of transparency:

Access: Hide differences in data representation and how a resource is accesed

Deals with data representation and the way resources can be accessed by users.

Location: Hide where a resource is located

Users shouldn't be able to tell where a resource is phisically located

Migration: Hide that a resource may move to another location

Users should be able to access a resource in the same way even if it has been moved

Relocation: Hide that a resource may be moved to another location while in use

Users shouldn't be able to notice that a resource they're accessing it being moved.

Replication: Hide that a resource is replicated

Deals with the fact that several copies of a resource exist

Concurrency: Hide that a resource may be shared by several competitive users

Users shouldn't be able to notice that other user is making use of the same resource

Failure: Hide a failure and recovery of a resource

Users do not notice that a resource fails to work properly AND the system recovers itself from that failure

Persistance: Hide whether a resource is in memory or on disk

Deals with masking whether a resource is in volatile memory or perhaps somewhere on a disk

Little endian -> high-order bytes are transmitted first

Big endian -> low-order bytes are transmitted first

Scalability measuring:

> Size: easily to add more resources and users

> Geographical: users and resources may lie far apart

> Administratively: easy to manage even if it spans many independent administrative organizations.

Scalability in one or more of these dimensions leads to performance loss.

SCALABILITY PROBLEMS

< Size: limitations of centralized services, data and algorithms.

< Decentralized algorithms:

- No machine has complete information about the system state

- Machines make decisions based onlu on local information

- Failure of one machine doesn't ruin the algorithm

- There's no implicit assumption that a global clock exists

< Greographical: Synchronous communication, unreliability, centralized solutions, conflicting policies

SCALABILITY TECHNIQUES

Solutions to scal. problems

> Hiding communication latencies: to not wait for responses to remote services requests ::Asynchronous communication:: other threads in the process can continue, when the reply comes in, the app is interrupted and a special handler is called to cmplete the previously issued request

If not possible, distribute the tasks in order to reduce the huge work load of a single server

> Distribution: Taking a component, splitting it into samaller parts and subsequently spreadint those parts across the system

> Replication: not only increases availability, but also helps to balance the load between components leading to better performance (hide communication latency)

» Caching: decision made by the client

--Leads to a consistency problems

Multiprocesors Multicomputers

Shared memory Homogeneous Heterogeneous

Single physical address Parallel systems

space that is shared by all CPUs

Bus -> there's a single network, backplanr, bus or other mediium that connects all the machines

Switched -> individual wires from machine to machine with many wiring patterns in use

## Multiprocesors ##

- All the CPUs have direct access to the shared memory

- Have a coherent memory

Bus gets overloaded immediately -[solution]-> add a high-speed cache memory betwee the CPU and the bus.

Cache stores the Most Recent Accessed Words. Cache size ~512kb to 1MB

Hit rate: probability of success

...But again, leads to coherence problems

and has limited scalability

The temporal solution for these problems are switches, two options are being study object:

Crossbar switch: the main advantage of these kind of switches is that every CPU can access any memory, and just when two CPUs try to access the same memory, one of them have to wait.

The bad news? It is expensive. For m processors and n memory units m\*n switches are required, which is absolutely everything but cheap.

Omega network: allow to connect every processor with every memory unit using quite fewer switches, the problem is that make the switching fast enough to ensure low latency is not cheap as well.

$$ Homogeneous Multicomputer Systems $$

Easier to be build than multiprocessors, have the same problem than them if connected on a bus-based connection: scalability, but whereas a multiprocessor losses performance with 5-6 CPUs, multicomputers lose that performance when the amount of nodes reach 25-100 (based upon the bandwith of the network), because the volume of traffic in a CPU-CPU connection is several orders of magnitud lower than when the interconnection network is also used for CPU-Memory.

Page 20 speaks about the different topologies used for implementing the switch based multicomputer and some differences between Massively Parallel Processors (MPPs) y Clusters of Workstations (COWs).

Nothing really interesting to note about.

%% Heterogenous multicomputer systems %%

The spectrum in which the computers that conforms an heterogenous MCS is as wide that some of their components may be high-performance parallel systems such as multiprocessors or homogeneous MCS.

Nevertheless, large scale heterogeneous MC lack a global system view, meaning they can't assume that the same performance or services are available everywhere, ergo sophisticated software is required to build applications for this kind of systems.

It is here where DS enter in action providing developers a layer of software that hides the underlying software, making a lot easier to develop applications for this complex hardware compositions

Software concepts

Software is what really makes distributed system a distributed system, it acts in two important ways:

> As resource manager for the underlying software allowing users and apps to share physical and virtual resources of all kinds.

> Hide intricacies and heterogeneous nature by providing a virtual machine where applications can be easily executed.

Operative systems

Can be divided in two groups:

- Tightly coupled OS: maintains a single, global view of resources it manages

> Distributed Operating System (DOS) - pretend hide the intricacies and underlying hardware

- Loosely coupled OS: better thought of 'em as a collection of computers each running their own OS that work together to make their own services and resources available to the others.

- Network Operating System (NOS) - offer local services to remote clients

Distributed operative systems

Most CPUs support at least two modes of operation:

Kernel mode: all instructions are permitted to be executed and the whole memory and collection of all registers is accessible during execution.

User mode: memory and register access is restricted.