DISTRIBUTED OPERATING SYSTEM

INTRODUCTION

An operating system is a program that controls the resources of a computer and provides its user with an interface or a virtual machine that's more convenient to use than bear machine.

To begin with, we use the term distributed system to mean a distributed operating system as opposed to a database system or some distributed application system such as a banking system, another name for a distributed operating system is DIS-**CENTRALIZED OPERATING SYSTEM.**
Example of a centralized (not distributed) operating system are; MS-DOS, UNIX, and CP/M.

A distributed operating system is the one that look to its user like an ordinary centralized operating system but runs on multiple, independent, central processing unit (CPU). The key concept in distributed operating system is the TRANPARENCY. What determine a distributed operating system are the software and not the hardware.

In distributed system, the error can be made to tolerate both hardware and software error but it is the software error and not the hardware that cleans the error when it occurs.

Network OS is used to manage Networked computer systems and create, maintain and transfer files in that Network.
Distributed OS is also similar to Networked OS but in addition to it the platform on which it is running should have high configuration such as more capacity RAM, High speed Processor. The main difference between the DOS and the NOS is the transparent issue: Transparency:

- How aware are users of the fact that multiple computers are being used?

Types of Distributed Operating Systems

- Network Operating Systems
- Distributed Operating Systems

Network-Operating Systems

- Users are aware where resources are located
- Network OS is built on top of centralized OS.
- Handles interfacing and coordination between local OSs.
- Users are aware of multiplicity of machines.

Distributed-Operating Systems

- Designed to control and optimize operations and resources in distributed system.
  - Users are not aware of multiplicity of machines
  - Access to remote resources similar to access to local resources
  - Data Migration – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task
  - Computation Migration – transfer the computation, rather than the data, across the system
- Computation speedup – sub processes can run concurrently on different sites
  - Process Migration – execute an entire process, or parts of it, at different sites
  - Load balancing – distribute processes across network to even the workload
  - Hardware preference – process execution may require specialized processor
  - Software preference – required software may be available at only a particular site
  - Data access – run process remotely, rather than transfer all data locally

EXAMPLES OF DISTRIBUTED OPERATING SYSTEM

- The Cambridge Distributed Computing System

- Amoeba
- The V Kernel
- The Eden Project

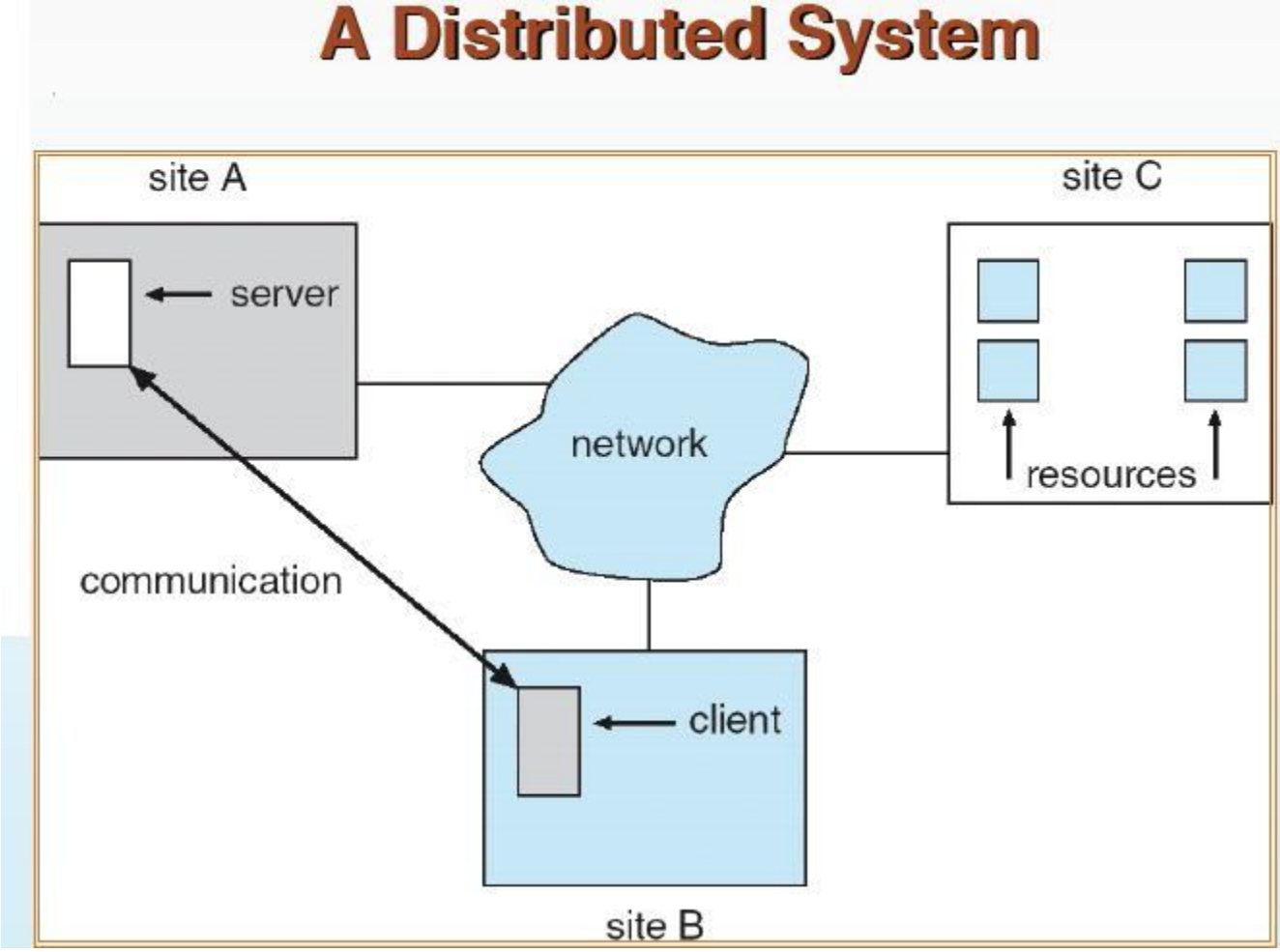
There are so many types of distributed operating system but these are chosen based on three criteria which are:

First, we only chose systems that were designed from scratch as-distributed systems (systems that gradually evolved by connecting together existing centralized systems or are multiprocessor versions of UNIX were excluded).

Second, we only chose systems that have actually been implemented; paper designs did not count.

Third, we only chose systems about which a reasonable amount of information was available.

NOTE: if a user can tell which computer he/she is using, then he/she is not using a distributed system. The user of a true distributed operating system should not know (or care) on which machine (or machines) their programs are running, where their files are stored and so on.



TRANSPARENCY

• Goal motivated by the desire to hide all irrelevant system-dependent details from the user, whenever possible.

• It is more important in distributed systems due to higher implementation complexities.

• **Shielding the system**-dependent information from the users is a trade-off between simplicity and effectiveness.

• **Access transparency** - Local and remote system entities must remain indistinguishable when viewed through the user interface. The distributed operating system maintains this perception through the exposure of a single access mechanism for a system entity, regardless of that entity being local or remote to the

user. Transparency dictates that any differences in methods of accessing any particular system entity—either local or remote—must be both invisible to, and undetectable by the user.

• **Location transparency** - Location transparency comprises two distinct sub-aspects of transparency, Naming transparency and User mobility. Naming transparency requires that nothing in the physical or logical references to any system entity should expose any indication of the entities location, or its local or remote relationship to the user. User mobility requires the consistent referencing of system entities, regardless of the system location from which the reference originates. Transparency dictates that the relative location of a system entity—either local or remote—must be both invisible to, and undetectable by the user.

• **Migration transparency** - Logical resources and physical processes migrated by the system, from one location to another in an attempt to maximize efficiency, reliability, availability, security, or whatever reason, should do so automatically controlled solely by the system. There are a myriad of possible reasons for migration; in any such event, the entire process of migration before, during, and after should occur without user knowledge or interaction. Transparency dictates that both the need for, and the execution of any system entity migration must be both invisible to, and undetectable by the user.

• **Concurrency transparency** - The distributed operating system allows for simultaneous use of system resources by multiple users and processes, which are kept completely unaware of the concurrent usage. Transparency dictates that both the necessity for concurrency and the multiplexed usage of system resources must be both invisible to, and undetectable by the user.

• **Replication transparency** - A system's elements or components may need to be copied to strategic remote points in the system in an effort to possibly increase efficiencies through better proximity, or provide for improved reliability through the duplication of a back-up. This duplication of a system entity and its subsequent movement to a remote system location may occur for any number of possible reasons; in any event, the entire process before, during, and after should occur without user knowledge or interaction. Transparency dictates that the necessity and execution of replication, as well as the existence of replicated entities throughout the system must be both invisible to, and undetectable by the user.

• **Parallelism transparency** - Arguably the most difficult aspect of transparency, and described by Tanenbaum as the "Holy grail" for distributed system designers. A system's parallel execution of a task among various processes throughout the system should occur without any required user knowledge or interaction. Transparency dictates that both the need for, and the execution of parallel processing must be both invisible to, and undetectable by the user.

• **Failure transparency** - In the event of a partial system failure, the system is responsible for the automatic, rapid, and accurate detection and orchestration of a remedy. These measures should exhibit minimal user imposition, and should initiate and execute without user knowledge or interaction. Transparency dictates that users and processes be exposed to absolute minimal imposition as a result of partial system failure; and any system-employed techniques of detection and recovery must be both invisible to, and undetectable by the user.

• **Performance transparency** - In any event where parts of the system experience significant delay or load imbalance, the system is responsible for the automatic, rapid, and accurate detection and orchestration of a remedy. These measures should exhibit minimal user imposition, and should initiate and execute without user knowledge or interaction. While reasonable and predictable performances are important goals in these situations, there should be no expressed or implied concepts of fairness or equality among affected users or processes. Transparency dictates that users and processes be exposed to absolute minimal imposition as a result of performance delay or load imbalance; and any system-employed techniques of detection and recovery must be both invisible to, and undetectable by the user.

• **Size transparency** - A system's geographic reach, number of nodes, level of node capability, or any changes therein should exists without any required user knowledge or interaction. Transparency dictates that system and node composition, quality, or changes to either must be both invisible to, and undetectable by the user.

• **Revision transparency** - System occasionally have need for system-software version changes and changes to internal implementation of system infrastructure. While a user may ultimately become aware of, or discover the availability of new system functions or services, their implementation should in no way be the prompt for this discovery. Transparency dictates that the implementation of system-software version changes and changes to internal system infrastructure must be

both invisible to, and undetectable by the user; except as revealed by administrators of the system.

SCHEDULLING TECHNIQUES

The hierarchical model provides a general model for resource control but does not provide any specific guidance on how to do scheduling. If each process uses an entire processor (i.e., no multiprogramming), and each process is independent of all the others, any process can be assigned to any processor at random. However, if it is common that several processes are working together and must communicate frequently with each other, as in UNIX pipelines or in cascaded (nested) remote procedure calls, then it is desirable to make sure that the whole group runs at once. Let us assume that each processor can handle up to N processes. If there are plenty of machines and N is reasonably large, the problem is not finding a free machine (i.e., a free slot in some process table), but something more subtle. The basic difficulty can be illustrated by an example in which processes A and B run on one machine and processes C and D run on another. Each machine is time shared in, say, 100-millisecond time slices, with A and C running in the even slices, and B and D running in the odd ones. Suppose that A sends many messages or makes many remote procedure calls to D. During time slice 0, A starts up and immediately calls D, which unfortunately is not running because it is now C's turn. After 100 milliseconds, process switching takes place, and D gets A's message, carries out the work, and quickly replies. Because B is now running, it will be another 100 milliseconds before A gets the reply and can proceed. The net result is one message exchange every 200 milliseconds. What is needed is a way to ensure that processes that communicate frequently run simultaneously. Although it is difficult to determine dynamically the inter-process communication patterns, in many cases a group of related processes will be started off together.

PROCESS MANAGEMENT

Process management provides policies and mechanisms for effective and efficient sharing of a system's distributed processing resources between that system's distributed processes. These policies and mechanisms support operations involving the allocation and de-allocation of processes and ports to processors, as well as provisions to run, suspend, migrate, halt, or resume execution of processes. While these distributed operating system resources and the operations on them can be either local or remote with respect to each other, the distributed operating system

must still maintain complete state of and synchronization over all processes in the system; and do so in a manner completely consistent from the user's unified system perspective.

As an example, load balancing is a common process management function. One consideration of load balancing is which process should be moved. The kernel may have several mechanisms, one of which might be priority-based choice. This mechanism in the kernel defines *what can be done*; in this case, choose a process based on some priority. The system management components would have policies implementing the decision making for this context. One of these policies would define what priority means, and how it is to be used to choose a process in this instance.

RESOURCES MANAGEMENT

Resource management in a distributed system differs from that in a centralized system in a fundamental way. Centralized systems always have tables that give complete and up-to-date status information about all the resources being managed; distributed systems do not. For example, the process manager in a traditional centralized operating system normally uses a “process table” with one entry per potential process. When a new process has to be started, it is simple enough to scan the whole table to see whether a slot is free. A distributed operating system, on the other hand, has a much harder job of finding out whether a processor is free, especially if the system designers have rejected the idea of having any central tables at all, for reasons of reliability. Furthermore, even if there is a central table, recent events on outlying processors may have made some table entries obsolete without the table manager knowing it. The problem of managing resources without having accurate global state information is very difficult.

PROCESSOR ALLOCATION

One of the key resources to be managed in a distributed system is the set of available processors. One approach that has been proposed for keeping tabs on a collection of processors is to organize them in a logical hierarchy independent of the physical structure of the network, as in MICROS. This approach organizes the machines like people in corporate, military, academic, and other real-world hierarchies. Some of the machines are workers and others are managers. For each group of k workers, one manager machine (the “department head”) is assigned the task of keeping track of who is busy and who is idle. If the system is large, there will be an unwieldy number of department heads; so some machines will function

as “deans,” riding herd on k department heads. If there are many deans, they too can be organized hierarchically, with a “big cheese” keeping tabs on k deans. This hierarchy can be extended ad infinitum, with the number of levels needed growing logarithmically with the number of workers. Since each processor need only maintain communication with one superior and k subordinates, the information stream is manageable. An obvious question is, “What happens when a department head, or worse yet, a big cheese, stops functioning (crashes)?” One answer is to promote one of the direct subordinates of the faulty manager to fill in for the boss. The choice of which one can either be made by the subordinates themselves, by the deceased’s peers, or in a more autocratic system, by the sick manager’s boss. To avoid having a single (vulnerable) manager at the top of the tree, one can truncate the tree at the top and have a committee as the ultimate authority. When a member of the ruling committee malfunctions, the remaining members promote someone one level down as a replacement. Although this scheme is not completely distributed, it is feasible and works well in practice. In particular, the system is self repairing, and can survive occasional crashes of both workers and managers without any long-term effects. In MICROS, the processors are monoprogrammed, so if a job requiring S processes suddenly appears, the system must allocate S processors for it. Jobs can be created at any level of the hierarchy. The strategy used is for each manager to keep track of approximately how many workers below it are available (possibly several levels below it). If it thinks that a sufficient number are available, it reserves some number R of them, where R 2 S, because the estimate of available workers may not be exact and some machines may be down. If the manager receiving the request thinks that it has too few processors available, it passes the request upward in the tree to its boss. If the boss cannot handle it either, the request continues propagating upward until it reaches a level that has enough available workers at its disposal. At that point, the manager splits the request into parts and parcels them out among the managers below it, which then do the same thing until the wave of scheduling requests hits bottom. At the bottom level, the processors are marked as “busy,” and the actual number of processors allocated is reported back up the tree. To make this strategy work well, R must be large enough so that the probability is high that enough workers will be found to handle the whole job. Otherwise, the request will have to move up one level in the tree and start all over, wasting considerable time and computing power. On the other hand, if R is too large, too many processors will be allocated, wasting computing capacity until word gets back to the top and they can be released. The whole situation is greatly complicated by the fact that requests for processors can be generated randomly anywhere in the system, so at any instant, multiple requests are likely to be in various stages of the allocation algorithm, potentially giving rise

to out-of-date estimates of available workers, race conditions, deadlocks, and more.

Failure Recovery

Failure Detection

Detecting hardware failure is difficult. To detect a link failure, a handshaking protocol can be used. Assume Site A and Site B has established a link. At fixed intervals, each site will exchange an I-am-up message indicating that they are up and running. If Site A does not receive a message within the fixed interval, it assumes either (a) the other site is not up or (b) the message was lost. Then, Site A can now send an "Are-you-up?" message to Site B. If Site A does not receive a reply, it can repeat the message or try an alternate route to Site B.
If Site A does not ultimately receive a reply from Site B, it concludes some type of failure has occurred in site B. Such failure could be that:
Types of failures:
- Site B is down
- The direct link between A and B is down
- The alternate link from A to B is down
- The message has been lost.

However, Site A cannot determine exactly **why** the failure has occurred

Reconfiguration

When Site A determines a failure has occurred, it must reconfigure the system:
1. If the link from A to B has failed, this must be broadcast to every site in the system
2. If a site has failed, every other site must also be notified indicating that the services offered by the failed site are no longer available.

When the link or the site becomes available again, this information must again be broadcast to all other sites.

Distributed Deadlock Detection

There are two kinds of potential deadlocks which are:

- resource deadlocks
- communication deadlocks

Resource deadlocks are traditional deadlocks, in which all of some set of processes are blocked waiting for resources held by other blocked processes. For example, if A holds X and B holds Y, and A wants Y and B wants X, a deadlock will result. In principle, this problem is the same in
Centralized and distributed systems, but it is harder to detect in the latter because there are no centralized tables.

The other kind of deadlock that can occur in a distributed system is a communication deadlock. Suppose A is waiting for a message from B and B is waiting for C and C is waiting for A. Then we have a deadlock. Chandy et al. [1983] present an algorithm for detecting (but not preventing) communication deadlocks. Very crudely summarized, they assume that each process that is blocked waiting for a message knows which process or processes might send the message. When a process logically blocks, they assume that it does not really block but instead sends a query message to each of the processes that might send it a real (data) message. If one of these processes is blocked, it sends query messages to the processes it is waiting for. If certain messages eventually come back to the original process, it can conclude that a deadlock exists. In effect, the algorithm is looking for a knot in a directed graph.

Redundancy Techniques

All the redundancy techniques that have emerged take advantage of the existence of multiple processors by duplicating critical processes on two or more machines. A particularly simple, but effective, technique is to provide every process with a backup process on a different processor. All processes communicate by message passing.

Whenever anyone sends a message to a process, it also sends the same message to the backup process. The system ensures that neither the primary nor the backup can continue running until it has been verified that both have correctly received the message. Thus, if one process crashes because of any hardware fault, the other one can continue. Furthermore, the remaining process can then clone itself, making a new backup to maintain the fault tolerance in the future. One disadvantage of duplicating every process is the extra processors required, but another, more subtle problem is that, if processes exchange messages at a high rate, a considerable amount of CPU time may go into keeping the processes synchronized at each message exchanged. If a process crashes, recovery is done by sending the most

recent checkpoint to an idle processor and telling it to start running. The recorder process then spoon feeds it all the messages that the original process received between the checkpoint and the crash. Messages sent by the newly restarted process are discarded. Once the new process has worked its way up to the point of crash, it begins sending and receiving messages normally, without help from the recording process.

STRENGTH AND WEAKNESS OF DISTRIBUTED OPERATING SYSTEM

STRENGTH

- The main goal of distributed system is the enormous rate of technological change in micro processor technology.
- Micro processors have become powerful and cheap compared with mainframes and minicomputer, so it has become attractive to think about designing large system that composes of many processors.
- Relative simplicity of software: each software has a dedicated function.
- Incremental growth.
- Reliability and availability.

WEAKNESS

- Unless one is very careful, it is easy for the communication protocol overhead to become a major source of in efficiency.
- With distributed systems, a high degree of fault tolerance is often, at least, an implicit goal.
- A more fundamental problem in distributed system is the lack of global state information.
- It is hard to schedule the processor optimally if you are not sure how many are up at the moment.

CONCLUSION

Distributed operating systems are still in an early phase of development, with many unanswered questions and relatively little agreement among workers in the field about how things should be done. Many experimental systems use the client-server model with some form of remote procedure call as the communication base, but there are also systems built on the connection model. Relatively little has been done on distributed naming, protection, and resource management, other than building straightforward name servers and process servers.

Fault tolerance is an up-and-coming area, with work progressing in redundancy techniques and atomic actions. Finally, a considerable amount of work has gone into the construction of file servers, print servers, and various other servers, but

here too there is much work to be done. The only conclusion that we draw is that distributed operating systems will be an interesting and fruitful area of research for a number of years to come.