# Distributed Operating System

#### A Comprehensive Review

Shahnawaz, Javed, Kaif

12321941, 12308921, 12326237

Lovely Professional University

Assistant professor: Dr. Anudeep Goraya

***Abstract- The paper aims at providing a complete survey of distributed operating systems. This paper also explain in detail various key design issues involved in the building of such system. A distributed operating system is a software over a collection of independent, networked, communicating, and physically separate computational nodes. Each individual node holds a specific software subset of the global aggregate operating system. Each subset is a composite of two distinct service provisioners. The first is a minimal kernel, or micro-kernel, that directly controls that node’s hardware. Second is a higher-level collection of system management components that coordinate the node's individual and collaborative activities. These components abstract micro-kernel functions and support user applications. The micro-kernel and the management components collection work together.***

1. INTRODUCTION

A distributed operating system is an operating system that runs on a number of technologies whose function is to make available a useful set of services, generally to make the set of machines act more like a only machine. A distributed OS provides the essential services and functionality required of an OS, adding attributes and particular configurations to allow it to support additional requirements such as increased scale and availability. To a user, a distributed OS works in a manner similar to a single-node, monolithic operating system. That is, although it consists of multiple nodes, it appears to users and applications as a single-node. Separating minimal system-level functionality from additional user-level modular services provides a “separation of mechanism and policy.”

* 1. HISTORY

Research and experimentation efforts began in earnest in the 1970s and continued through 1990s, with focused interest peaking in the late 1980s. A number of distributed operating systems were introduced during this period; however, very few of these implementations achieved even modest commercial success. Fundamental and pioneering implementations of primitive distributed operating system component

concepts date to the early 1950s.Some of these individual steps were not focused directly on distributed computing, and at the time, many may not have realized their important impact. These pioneering efforts laid important groundwork, and inspired continued research in areas related to distributed computing. In the mid-1970s, research produced important advances in distributed computing. These breakthroughs provided a solid, stable foundation for efforts that continued through the 1990s.

* 1. EXAMPLES OF DISTRIBUTED OPERATING SYSTEMS

l. IRIX operating system; is the implementation of UNIX System V, Release 3 for Silicon Graphics multiprocessor workstations.

2. AIX operating system for IBM RS/6000 computers.

3. Solaris operating system for SUN multiprocessor workstations.

1.3. GOALS

1. There are many different types of distributed computing systems and many challenges to overcome in successfully designing one. The main goal of a distributed computing system is to connect users and resources in a transparent, open, and scalable way.
2. One design goal in building a distributed system is to create a single system image; to have a collection of independent computers appear as a single system to the user.
3. ASPECTS OF DISTRIBUTED OPERATING SYSTEM

Although all Distributed Systems consist of multiple CPUs, there are different ways of interconnecting them and how they communicate Flynn (1972) identified two essential characteristics to classify multiple CPU computer systems: the number of instruction streams and the number of data streams

1. **Uniprocessors SISD**
2. **Array processors are SIMD** - processors cooperate on a single problem
3. **MISD** - No known computer fits this model
4. **Distributed Systems are MIMD** MIMD can be split into two classifications

**Tightly-coupled** - short delay in communication between computers, high data rate (e.g., Parallel computers working on related computations)(MULTIPROCESSORS)

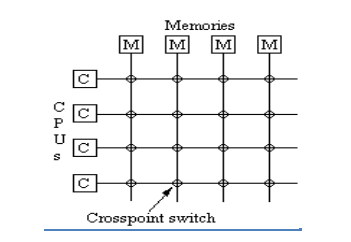
**Loosely-coupled** - Large delay in communications, Low data rate (Distributed Systems working on unrelated computations)(MULTICOMPUTERS) Can be further subclassified as **Bus** - All machines connected by single medium (e.g., LAN, bus, backplane, cable)

**Switched** - Single wire from machine to machine, with possibly different wiring patterns (e.g, Internet)

**MULTIPROCESSORS:** CPUS have shared memory.

1. **Bus-based multiprocessors**: In a bus-based system, all CPUs are connected to one system bus. System memory and peripherals are also connected to that bus. If CPU A writes a word to a memory location and CPU B can read that same word.

**Crossbar switch**:



**MULTICOMPUTERS**: CPUs have separate memories.

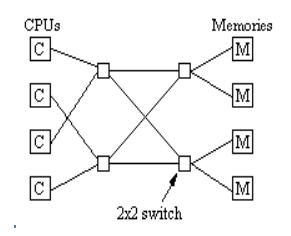
1. **Bus-based multi-computers**: Bus-based multi-computers are easier to design in that we don’t need to contend with issues of shared memory: every CPU simply has its own local memory. The communication network between the two is a bus (for example, an Ethernet local area network).
2. **Switched Multicomputers**: With a switched interconnect, all hosts connect to a network switch. The switch moves traffic only back at any time after the write, the memory is coherent.

Bus-based interconnect with cache A bus can get overloaded rather quickly with each CPU accessing the bus for all of its instructions and data. A solution to this is to add cache memory between the CPU and the bus .

1. **Switched Multiprocessors** 

* Used for more than 64 CPUs 
* Split memory into smaller modules 
* Connect all CPUs to each memory module, two common method between communicating hosts, allowing other hosts to communicate without seeing their network speeds diminish. The huge benefit of switching is that it gives us a scalable network.

**Omega network:**



2.2) **SOFTWARE CONCEPTS :**

There is no single definition or goal of distributed software but in designing distributed software, we often touch upon the same set of goals and problems. These general goals are transparency and scalability. This covers problems as diverse as: 

* A network of redundant web servers
* Thousands of machines participating together in processing your search query.

The next step in building distributed systems is placing tightly-coupled software on loosely-coupled hardware. With this structure we attempt to make a network of machines appear as one single timesharing system, realizing the single system image. Users should not be aware of the fact that the machine is distributed and contains multiple CPUs. If we succeed in this, we will have a true distributed system.

1. TYPES OF DISTRIBUTED OPERATING SYSTEM

**Network Operating Systems :**

* Loosely-coupled software on loosely-coupled hardware
* E.g., LAN with file server
* Users are aware that they are using independent hardware, but share a consistent view of the filing system with other network users

**True Distributed Systems:**

* Tightly-coupled software on Loosely-coupled hardware
* Give users impression that collection of computers is a single time-sharing system -the virtual uni-processor
* Processes are capable of being executed on any computer on the network, and users won't realise.
* **Multiprocessor Timesharing Systems:**  Tightly-coupled software on toghtly-coupled hardware 
* Single RUN Queue 
* Scheduler must run as a critical section to prevent two CPUs from selecting the same process to run

1. **ADVANTAGES OVER CENTRALIZED SYSTEMS**
2. **Speed:** When used to implement parallel processing where only goal is to achieve maximum speed on a single problem, distributed systems can achieve very high specd as compared to the centralized ones.
3. **Inherent Distribution:** Another reason for building a distributed system is that some applications are inherently distributed. Banking, Airline reservation etc. are examples of the applications that are inherently distribute .
4. **Reliability:** one is higher reliability. By distributing the workload over many machines, a single chip failure will bring down at most one machine, leaving the rest intact. For critical applications, such as control of nuclear reactors or aircraft, using a distributed system to achieve high reliability may be a dominant consideration. 4. **Incremental Growth:** It may be possible to simply add more processors to the system, thus allowing it to expand gradually as the need arises.
5. **DESIGN ISSUES**
6. **Transparency**: At the high levels, transparency means hiding distribution from the users. At the low levels, transparency means hiding the distribution from the programs. There are several forms of transparency:
7. **Location transparency** : Users don’t care where the resources are located. Migration transparency : Resources may move at will. **(B)Replication transparency** : Users cannot tell whether there are multiple copies of the same resource.
8. **Concurrency transparency**: Users share resources transparently with each other without interference.
9. **Parallelism transparency**: Operations can take place in parallel without the users knowing. **2. Flexibility**: It should be easy to develop distributed systems. One popular approach is through the use of a microkernel. A microkernel is a departure from the monolithic operating systems that try to handle all system requests. Instead, it supports only the very basic operations: IPC, some memory management, a small amount of process management, and low-level I/O. All else is performed by user-level servers.

**3.Reliability**:. Reliability encompasses a few factors: data must not get lost, the system must be secure, and the system must be fault tolerant.

1. **Performance**:. The communication links may be slow and affect network performance. If we exploit parallelism, it may be on a fine grain (within a procedure, array ops, etc.) or a coarse grain (procedure level, service level).
2. **Scalability** : We’d like a distributed system to scale indefinitely. This generally won’t be possible, but the extent of scalability will always be a consideration.

**VI. RESOURCE 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. The problem of managing resources without having accurate global state information is very difficult. Relatively little work has been done in this area. In the following sections we will look at some work that has been done on distributed process

management and scheduling.

**6.1. Process Management**

One of the key resources to be managed in a distributed system is the set of available processors.If these machines are multi-programmed (time-shared), with N process table slots per machine, each physical

processor can be regarded as N virtual processors to be managed. 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 [Wittie and van Tilborg 1980]. 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, 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 gives up the ghost, the remaining members promote someone one level down as replacement.

While this scheme is not completely distributed, it is feasible, and in practice works well. In particular, the system is self-repairing, and can survive occasional crashes of both workers and managers without any long-term effects.

**6.2. Scheduling**

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 multi-programming), 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 the whole group runs at once. We will first look at one scheduling strategy proposed for mono-programmed systems, and then at one for multi-programmed systems.

In MICROS the processors are not multi-programmed, 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 described earlier. 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 >= 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 upwards 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 chosen large enough 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 chosen too large, too many processors will be allocated, wasting computing capacity until word gets

back to the top and they can be released. In [Van Tilborg and Wittie 1981] a mathematical analysis of the problem is given and various other aspects not described here are covered in detail.

Ousterhout [1982] has proposed several algorithms based on the concept of co-scheduling which takes interprocess communication patterns into account while scheduling to ensure that all members of a

group run at the same time. The first algorithm uses a conceptual matrix in which each column is the process table for one machine. Thus column 4 consists of all the processes that run on machine 4. Row 3 is the collection of all processes that are in slot 3 of some machine, starting with the process in slot 3 of machine 0, then the process in slot 3 of machine 1, and so on. The gist of his idea is to have each processor use a round robin scheduling algorithm with all processors first running the process in slot 0 for a fixed period, then all processors running the process in slot 1 for a fixed period, etc. A broadcast message could be used to tell each processor when to do process switching, to keep the time slices synchronized.

By putting all the members of a process group in the same slot number, but on different machines, one has the advantage of N-fold parallelism, with a guarantee that all the processes will be run at the same time, to maximize communication throughput. This scheduling technique can be combined with the

hierarchical model of process management used in MICROS by having each department head maintain the matrix for its workers, assigning processes to slots in the matrix and broadcasting time signals.

Ousterhout also described several variations to this basic method to improve performance. One of these breaks the matrix into rows, and concatenates the rows to form one long row. With k machines, any k consecutive slots belong to different machines. To allocate a new process group to slots, one lays a window k slots wide over the long row such that the leftmost slot is empty but the slot just outside the left edge

of the window is full. If sufficient empty slots are present in the window, the processes are assigned to the empty slots, otherwise the window is slid to the right and the algorithm repeated. Scheduling is done by starting the window at the left edge and moving rightward by about one window’s worth per time slice,

taking care not to split groups over windows. Ousterhout’s paper discusses these and other methods in more detail and gives some performance results.

A completely different approach to scheduling that does not require any tables at all is bidding

[Farber and Larson 1972]. When a process (e.g., a command interpreter) has a need to create a new process in order to run some program, it broadcasts a requests for bids on the network, describing what kind of processor it needs or what program it wants run. All available processors are normally enabled to listen

for such requests. When an available processor sees a request-for-bid message, it checks to see if it has the appropriate configuration (e.g., floating point hardware, enough memory), and if so, makes a bid. The requesting process evaluates the bids and then selects one or more bidders to run the new process(es).

This strategy can be generalized to multi-programmed processors by having the bids tell how much CPU capacity the bidder is willing to divert to the new process if it wins the bid. A heavily loaded processor will obviously be able to devote less capacity to a new process than a lightly loaded one. Therefore lightly loaded processors will usually win until they get too much work, at which time they will begin losing. This scheme thus provides for an automatic, and fully dynamic, form of load balancing.

**VII. FAULT TOLERANCE**

In the past few years, two approaches to making distributed systems fault tolerant have emerged.

They differ radically in orientation, goals, and attitude toward the theologically sensitive issue of the perfect-ability of mankind (programmers in particular). One approach is based on redundancy and one is based on the notion of an atomic transaction. Both are described briefly below.

**7.1. Redundancy Techniques**

All the redundancy techniques 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 insures that neither the primary nor the backup can continue running until it has been verified that both have correctly received the message.

The fault tolerance comes from the property that if one process crashes due to any hardware fault, the other one can continue. Furthermore, the remaining process can then clone itself, thus making a new backup to maintain the fault tolerance in the future. Borg et al. [1983] have described a system using these principles.

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 exchange. Powell and Presotto [1983]

have described a redundant system that puts almost no additional load on the processes being backed up.

In their system, all messages sent on the network are recorded by a special "recorder" process. From time to time, each process checkpoints itself onto a remote disk.

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.

Both of the above techniques are only applicable to making the system tolerant of hardware errors.

However, it is also possible to use the redundancy inherent in distributed systems to make systems tolerant of software errors as well. One approach is to structure each program as a collection of modules, each one with a well-defined function and a very precisely specified interface with all the others. Instead of writing

a module only once, N programmers are asked to program it, yielding N functionally identical modules.

During execution, the program runs on N machines in parallel. After each module finishes, the machines compare their results and vote on the answer. If a majority of the machines say that the answer is X , then all of them use X as the answer, and all continue in parallel with the next module. In this manner, the effects of an occasional software bug can be voted down. If formal specifications for any of the modules are available, the answers can also be checked against the specifications to guard against the

possibility of accepting an answer that is clearly wrong. Some work in this area is discussed in [Avizienis and Kelly 1984]

**7.2. Atomic Transactions**

When multiple users spread over several machines are concurrently updating a distributed data base, and one or more machines crash, the potential for chaos is truly impressive. In a certain sense, the current situation is a step backwards from the technology of the 1950s. The normal way of updating a data base then was to have one magnetic tape, called the "master file," and one or more tapes with updates (e.g., daily sales reports from all of a company’s stores). The master tape and the updates were brought to the computer center, which then mounted the master tape and one update tape, and ran the update program to produce a new master tape. This new tape was then used as the "master" for use with the next update tape.

This scheme had the very real advantage that if the update program crashed, one could always fall back on the previous master tape and the update tapes. In other words, an update run could be viewed as either running correctly to completion (and producing a new master tape), or having no effect at all (crash part way through, new tape discarded). Furthermore, update jobs from different sources always ran in

some (undefined) sequential order. It never happened that two concurrent users would read a field in a record, (e.g. 6), each add 1 to the value, and then each store a 7 into that field, instead of the first one storing a 7 and the second one storing an 8.The property of run-to-completion or do-nothing is called an atomic update. The property of notinterleaving two jobs is called serializability. The goal of the people working on the atomic transaction

approach to fault tolerance has been to regain the advantages of the old tape system without giving up the convenience of data bases on disk that can be modified in place, and furthermore to be able to do everything in a distributed way.

Lampson [1981] has described a way of achieving atomic transactions by building up a hierarchy of abstractions. We will summarize his model below. Real disks can crash during READ and WRITE operations in unpredictable ways. Furthermore, even if a disk block is correctly written, there is a small, but nonzero probability of it subsequently being corrupted by newly developed bad spot on the disk surface.

The model assumes that spontaneous block corruptions are sufficiently infrequent that the probability of two such events happening within some predetermined time, T , is negligible. To deal with real disks, the system software must be able to tell if a block is valid or not, for example, by using a checksum.

The first layer of abstraction on top of the real disk is the "careful disk," in which every

CAREFUL-WRITE is read back immediately to verify that it is correct. If the CAREFUL-WRITE persistently fails, the system marks the block as "bad" and then intentionally crashes. Since **Careful-writes** are verified, **Careful-reads** will always be good, unless a block has gone bad after being written and verified.

The next layer of abstraction is stable storage. A stable storage block consists of an ordered pair of careful blocks, typically corresponding blocks on different drives to minimize the chance of both being damaged by a hardware failure. The stable storage algorithm guarantees that at least one of the blocks is always valid. The STABLE-WRITE primitive first does a CAREFUL-WRITE on one block of the pair, and then the other. If the first one fails, a crash is forced, as mentioned above, and the second one is left

untouched.

After every crash, and at least once every time T , a special cleanup process is run to examine each stable block. If both blocks are "good" and identical, nothing has to be done. If one is "good" and one is "bad" (failure during a CAREFUL-WRITE), the "bad" one is replaced by the "good" one. If both are "good" but different (crash between two **Careful-writes**), the second one is replaced by a copy of the first one. This algorithm allows individual disk blocks to be updated atomically and survive infrequent crashes.

Stable storage can be used to create "stable processors" [Lampson 1981]. To make itself crashproof, a CPU must checkpoint itself on stable storage periodically. If it subsequently crashes, it can always restart itself from the last checkpoint. With stable storage and stable processors, processes can implement atomic transactions by first writing an intentions list to stable storage and then writing a commit flag to

stable storage.

**VIII. APPLICATIONS**

**Network applications:**

o World wide web and peer-to-peer networks

o Massively multiplayer online games and virtual reality communities

o Distributed databases and distributed database management systems

o Distributed information processing systems such as banking systems and airline reservation systems 

**Real-time process control:**

o Aircraft control systems

o Industrial control systems 

**Parallel computation:**

o Scientific computing, including cluster computing and grid computing and various volunteer computing project.

**IX.CONCLUSION**

Distributed systems consist of independent CPUs that work together to make the absolute system look like a single computer. They have a number of possible selling points, including good price/performance ratios, the capability to match distributed applications well, potentially high consistency and incremental increase as the workload grows. They also have some disadvantages, such as extra composite software, potential communication bottlenecks, and weak security. Nevertheless, there is significant interest worldwide in building and installing them.. Distributed operating systems turn the entire set of hardware and software into a single integrated system, much like a usual timesharing system. Distributed systems have to be designed suspiciously. A key topic is simplicity — hiding all the distribution from the users and still from the application programs. Another issue is flexibility. the design should be made with the scheme of making potential changes easy.

***REFERENCES***

1. *Dos concepts and design By Pradeep K. Sinha*

*[2] Dos concepts and design ByAndrew S.Tanenbaum*

*[3] Distributed Systems: Principles and Paradigms Hardcover, By Andrew S. Tanenbaum (Author), Maarten van Steen (Author)*