# TOPIC 2

# ISSUES IN DESIGNING A DISTRIBUTED OPERATING SYSTEM

## **Introduction**

Over the past two decades, advancements in microelectronic technology have resulted in the availability of fast, inexpensive processors, and advancements in communication technology have resulted in the availability of cost effective and highly efficient computer networks. This Topic will enhance understand what is meant by Distributed systems.

## **Objectives**

By the end of the course, students should understand: -

2.1 Issues in Designing a Distributed Operating System

2.2 Transparency

2.3 Performance Transparency

2.4 Scaling Transparency

2.5 Reliability

2.6 Fault Avoidance

2.7 Fault Tolerance

2.8 Fault Detection and Recovery

2.9 Flexibility

2.10 Performance

2.11 Scalability

**Learning Activity 2.0**

1.Differentiate between Fault Tolerance and Fault Avoidance

2. Discuss Scaling Transparency

**2.1 2.1 ISSUES IN DESIGNING A DISTRIBUTED OPERATING SYSTEM**

In general, designing a distributed operating system is more difficult than designing a centralized operating system for several reasons.

* In the design of a centralized operating system, it is assumed that the operating system has access to complete and accurate information about the environment in which it is functioning. For example, a centralized operating system can request status information, being assured that the interrogated component will not charge state while awaiting a decision based on that status information, since only the single operating system asking the question may give commands. However, a distributed operating system must be designed with the assumption that complete information about the system environment will never be available.
* In a distributed system, the resources are physically separated, there is no common clock among the multiple processors, delivery of messages is delayed, and messages could even be lost. Due to all these reasons, a distributed operating system does not have up-to-date, consistent knowledge about the state of the various components of the underlying distributed system. Obviously, lack of up-to-date and consistent information makes many things (Such as management of resources and synchronization of cooperating activities) much harder in the design of a distributed operating system. For example, it is hard to schedule the processors optimally if the operation system is not sure how many of them are up at the moment.

Despite these complexities and difficulties, a distributed operating system must be designed to provide all the advantages of a distributed system to its users. That is, the users should be able to view a distributed system as a virtual centralized system that is flexible, efficient, reliable, secure and easy to use. To meet this challenge, the designers of a distributed operating system must deal with several design issues

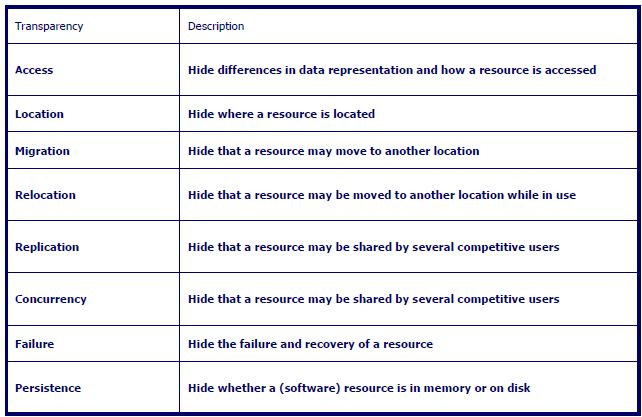
**Design Issues**

**2.2 TRANSPARENCY**

A distributed system that is able to present itself to user and application as if it were only a single computer system is said to be **transparent**.

There are eight types of transparencies in a distributed system:

1. **Access Transparency:** It hides differences in datarepresentation and how a resource is accessed by a user. Example, a distributed system may have a computer system that runs different operating systems, each having their own file naming conventions. Differences in naming conventions as well as how files can be manipulated should be hidden from the users and applications.
2. **Location Transparency:** Hides where exactly the resource islocated physically. Example, by assigning logical names to resources like yahoo.com, one cannot get an idea of the location of the web page’s main server.
3. **Migration Transparency:** Distributed system in whichresources can be moved without affecting how the resource can be accessed are said to provide migration transparency. It hides that the resource may move from one location to another.
4. **Relocation Transparency:** this transparency deals with the factthat resources can be relocated while it is being accessed without the user who is using the application to know anything. Example: using a Wi-Fi system on laptops while moving from place to place without getting disconnected.
5. **Replication Transparency:** Hides the fact that multiple copiesof a resource could exist simultaneously. To hide replication, it is essential that the replicas have the same name. Consequently, as system that supports replication should also support location transparency.
6. **Concurrency Transparency:** It hides the fact that the resourcemay be shared by several competitive users. Example, two independent users may each have stored their file on the same server and may be accessing the same table in a shared database. In such cases, it is important that each user doesn’t notice that the others are making use of the same resource.
7. **Failure Transparency:** Hides failure and recovery of theresources. It is the most difficult task of a distributed system and is even impossible when certain apparently realistic assumptions are made. Example: A user cannot distinguish between a very slow or dead resource. Same error message come when a server is down or when the network is overloaded of when the connection from the client side is lost. So here, the user is unable to understand what has to be done, either the user should wait for the network to clear up, or try again later when the server is working again.
8. **Persistence Transparency:** It hides if the resource is inmemory or disk. Example, Object oriented database provides facilities for directly invoking methods on storage objects. First the database server copies the object states from the disk i.e. main memory performs the operation and writes the state back to the disk. The user does not know that the server is moving between primary and secondary memory.



**Summary of the transparencies**

In a distributed system, multiple users who are spatially separated use the system concurrently. In such a duration, it is economical to share the system resources (hardware or software) among the concurrently executing user processes. However since the number of available resources in a computing system is restricted, one user process must necessarily influence the action of other concurrently executing user processes, as it competes for resources. For example, concurrent updates to the same file by two different processes should be prevented. Concurrency transparency means that each user has a feeling that he or she is the sole user of the system and other users do not exist in the system. For providing concurrency transparency, the resource sharing mechanisms of the distributed operating system must have the following four properties :

1. An event-ordering property ensures that all access requests to various system resources are properly ordered to provide a consistent view to all users of the system.
2. A mutual-exclusion property ensures that at any time at most one process accesses a shared resource, which must not be used simultaneously by multiple processes if program operation is to be correct.
3. A no-starvation property ensures that if every process that is granted a resource, which must not be used simultaneously by multiple processes, eventually releases it, every request for that resource is eventually granted.
4. A no-deadlock property ensures that a situation will never occur in which competing processes prevent their mutual progress even though no single one requests more resources than available in the system.

**2.3 PERFORMANCE TRANSPARENCY**

The aim of performance transparency is to allow the system to be automatically reconfigured to improve performance, as loads vary dynamically in the system. As far as practicable, a situation in which one processor of the system is overloaded with jobs while another processor is idle should not be allowed to occur. That is, the processing capability of the system should be uniformly distributed among the currently available jobs in the system.

This requirement calls for the support of intelligent resource allocation and process migration facilities in distributed operating systems.

**2.4 SCALING TRANSPARENCY**

The aim of scaling transparency is to allow the system to expand in scale without disrupting the activities of the users. This requirement calls for open-system architecture and the use of scalable algorithms for designing the distributed operating system components**.**

**2.5 RELIABILITY**

In general, distributed systems are expected to be more reliable than centralized systems due to the existence of multiple instances of resources. However, the existence of multiple instances of the resources alone cannot increase the system’s reliability. Rather, the distributed operating system, which manages these resources must be designed properly to increase the system’s reliability by taking full advantage of this characteristic feature of a distributed system.

A fault is a mechanical or algorithmic defect that may generate an error. A fault in a system causes system failure. Depending on the manner in which a failed system behaves, system failures are of two types –

* **Fail-stop failure** [Schlichting and Schneider 1983] - the system stops functioning after changing to a state in which its failure can be detected
* **Byzantine failure** [Lamport et al. 1982] - the system continues to function but produces wrong results.

Undetected software bugs often cause Byzantine failure of a system. Obviously, Byzantine failures are much more difficult to deal with than fail-stop failures.

For higher reliability, the fault-handling mechanisms of a distributed operating system must be designed properly to avoid faults, to tolerate faults, and to detect and recover form faults. Commonly used methods for dealing with these issues are briefly described text.

**2.6 FAULT AVOIDANCE**

Fault avoidance deals with designing the components of the system in such a way that the occurrence of faults in minimized. Conservative design practice such as using high reliability components are often employed for improving the system’s reliability based on the idea of fault avoidance. Although a distributed operating system often has little or no role to play in improving the fault avoidance capability of a hardware component, the designers of the various software components of the distributed operating system must test them thoroughly to make these components highly reliable.

**2.7 FAULT TOLERANCE**

Fault tolerance is the ability of a system to continue functioning in the event of partial system failure. The performance of the system might be degraded due to partial failure, but otherwise the system functions properly. Some of the important concepts that may be used to improve the fault tolerance ability of a distributed operating system are as follows:

1. **Redundancy techniques:** The basic idea behind redundancytechniques is to avoid single points of failure by replicating critical hardware and software components, so that if one of them fails, the others can be used to continue. Obviously, having two or more copies of a critical component makes it possible, at least in principle, to continue operations in spite of occasional partial failures. For example, a critical process can be simultaneously executed on two nodes so that if one of the two nodes fails, the execution of the process can be completed at the other node. Similarly, a critical file may be replicated on two or more storage devices for better reliability.

Notice that with redundancy techniques additional system overhead is needed to maintain two or more copies of a replicated resource and to keep all the copies of a resource consistent. For example, if a file is replicated on two or more nodes of a distributed system, additional disk storage space is required and for correct functioning, it is often necessary that all the copies of the file are mutually consistent. In general, the larger is the number of copies kept, the better is the reliability but the incurred overhead involved. Therefore, a distributed operating system must be designed to maintain a proper balance between the degree of reliability and the incurred overhead. This raises an important question : How much replication is enough? For an answer to this question, note that a system is said to be k-fault tolerant if it can continue to function even in the event of the failure of k components [Cristian 1991, Nelson 1990]. Therefore, if the system is to be designed to tolerance k fail – stop failures, k + 1 replicas are needed. If k replicas are lost due to failures, the remaining one replica can be used for continued functioning of the system. On the other hand, if the system is to be designed to tolerance k Byzantine failures, a minimum of 2k + 1 replicas are needed. This is because a voting mechanism can be used to believe the majority k + 1 of the replicas when k replicas behave abnormally.

Another application of redundancy technique is in the design of a stable storage device, which is a virtual storage device that can even withstand transient I/O faults and decay of the storage media. The reliability of a critical file may be improved by storing it on a stable storage device.

1. **Distributed control:** For better reliability, many of the particularalgorithms or protocols used in a distributed operating system must employ a distributed control mechanism to avoid single points of failure. For example, a highly available distributed file system should have multiple and independent file servers controlling multiple and independent storage devices. In addition to file servers, a distributed control technique could also be used for name servers, scheduling algorithms, and other executive control functions. It is important to note here that when multiple distributed servers are used in a distributed system to provide a particular type of service, the servers must be independent. That is, the design must not require simultaneous functioning of the servers; otherwise, the reliability will become worse instead of getting better.

**2.8 FAULT DETECTION AND RECOVERY**

The faulty detection and recovery method of improving reliability deals with the use of hardware and software mechanisms to determine the occurrence of a failure and then to correct the system to a state acceptable for continued operation.

Some of the commonly used techniques for implementing this method in a distributed operating system are as follows.

1. **Atomic transactions:** An atomic transaction (or justtransaction for shore) is a computation consisting of a collection of operation that take place indivisibly in the presence of failures and concurrent computations. That is, either all of the operations are performed successfully or none of their effects prevails, other processes executing concurrently cannot modify or observe intermediate states of the computation. Transactions help to preserve the consistency of a set of shared date objects (e.g. files) in the face of failures and concurrent access. They make crash recovery much easier, because transactions can only end in two states : Either all the operations of the transaction are performed or none of the operations of the transaction is performed.

In a system with transaction facility, if a process halts unexpectedly due to a hardware error before a transaction is completed, the system subsequently restores any data objects that were undergoing modification to their original states. Notice that if a system does not support a transaction mechanism, unexpected failure of a process during the processing of an operation may leave the data objects that were undergoing modification in an inconsistent state. Therefore, without transaction facility, it may be difficult or even impossible in some cases to roll back (recover) the data objects from their current inconsistent states to their original states.

1. **Stateless servers:** The client-server model is frequently used indistributed systems to service user requests. In this model, a server may be implemented by using any one of the following two service paradigms – **stateful** or **stateless**. The two paradigms are distinguished by one aspect of the client – server relationship, whether or not the history of the serviced requests between a client and a server affects the execution of the next service request. The stateful approach does depend on the history of the serviced requests, but the stateless approach does not depend on it. Stateless servers have a distinct advantage over stateful servers in the event of a failure. That is, the stateless service paradigm makes crash recovery very easy because no client state information is maintained by the server. On the other hand, the stateful service paradigm requires complex crash recovery procedures. Both the client and server need to reliably detect crashes. The server needs to detect client crashes so that it can discard any state it is holding for the client, and the client must detect server crashes so that it can perform necessary error – handling activities. Although stateful service becomes necessary in some cases, to simplify the failure detection and recovery actions, the stateless service paradigm must be used, wherever possible.
2. Acknowledgments and timeout-based retransmission of messages. In a distributed system, events such as a node crash or a communication link failure may interrupt a communication that was in progress between two processes, resulting in the loss of a message. Therefore, a reliable interprocess communication mechanism must have ways to detect lost messages so that they can be retransmitted. Handling of lost messages usually involves return of acknowledgment messages and retransmissions on the basis of timeouts. That is, the receiver must return an acknowledgment message for every message received, and if the sender does not receive any acknowledgement for a message within a fixed timeout period, it assumes that the message was lost and retransmits the message. A problem associated with this approach is that of duplicate message. Duplicates messages may be sent in the event of failures or because of timeouts. Therefore, a reliable interprocess communication mechanism should also be capable of detecting and handling duplicate messages. Handling of duplicate messages usually involves a mechanism for automatically generating and assigning appropriate sequence numbers to messages. Use of acknowledgement messages, timeout-based retransmissions of messages, and handling of duplicate request messages for reliable communication.

The mechanisms described above may be employed to create a very reliable distributed system. However, the main drawback of increased system reliability is potential loss of execution time efficiency due to the extra overhead involved in these techniques. For many systems it is just too costly to incorporate a large number of reliability mechanisms. Therefore, the major challenge for distributed operating system designers is to integrate these mechanisms in a cost-effective manner for producing a reliable system.

**2.9 FLEXIBILITY**

Flexibility is the most important features for open distributed systems. The design of a distributed operating system should be flexible due to the following reasons:

1. **Ease of modification:** From the experience of systemdesigners, it has been found that some parts of the design often need to be replaced / modified either because some bug is detected in the design or because the design is no longer suitable for the changed system environment or new-user requirements. Therefore, it should be easy to incorporate changes in the system in a user-transparent manner or with minimum interruption caused to the users.
2. **Ease of enhancement:** In every system, new functionalitieshave to be added from time to time it more powerful and easy to use. Therefore, it should be easy to add new services to the system. Furthermore, if a group of users do not like the style in which a particular service is provided by the operating system, they should have the flexibility to add and use their own service that works in the style with which the users of that group are more familiar and feel more comfortable.

The most important design factor that influences the flexibility of a distributed operating system is the model used for designing its kernel. The kernel of an operating system is its central controlling part that provides basic system facilities. It operates in a separate address space that is inaccessible to user processes. It is the only part of an operating system that a user cannot replace or modify. We saw that in the case of a distributed operating system identical kernels are run on all the nodes of the distributed system.

The two commonly used models for kernel design in distributed operating systems are the **monolithic kernel** and the **microkernel**. In the monolithic kernel model, most operating system services such as process management, memory management, device management, file management, name management, and inter-process communication are provided by the kernel. As a result, the kernel has a large, monolithic structure. Many distributed operating systems that are extensions or limitations of the UNIX operating system use the monolithic kernel model. This is mainly because UNIX itself has a large, monolithic kernel.

On the other hand, in the microkernel model, the main goal is to keep the kernel as small as possible. Therefore, in this model, the kernel is a very small nucleus of software that provides only the minimal facilities necessary for implementing additional operating system services. The only services provided by the kernel in this model are inter-process communication low level device management, a limited amount of low-level process management and some memory management. All other operating system services, such as file management, name management, additional process, and memory management activities and much system call handling are implemented as user-level server processes. Each server process has its own address space and can be programmed separately.

As compared to the monolithic kernel model, the microkernel model has several advantages. In the monolithic kernel model, the large size of the kernel reduces the overall flexibility and configurability of the resulting operating system. On the other hand, the resulting operating system of the microkernel model is highly modular in nature. Due to this characteristic feature, the operating system of the microkernel model is easy to design, implement, and install. Moreover, since most of the services are implemented as user-level server processes, it is also easy to modify the design or add new services.

**24**

In spite of its potential performance cost, the microkernel model is being preferred for the design of modern distributed operating systems. The two main reasons for this are as follows.

1. The advantages of the microkernel model more than compensate for the performance cost. Notice that the situation here is very similar to the one that caused high level programming languages to be preferred to assembly languages. In spite of the better performance of programs written in assembly languages, most programs are written in high-level languages due to the advantages of ease of design, maintenance, and portability. Similarly, the flexibility advantages of the microkernel model previously described more than outweigh its small performance penalty.
2. Some experimental results have shown that although in theory the microkernel model seems to have poorer performance than the monolithic kernel model, this is not true in practice. This is because other factors tend to dominate, and the small overhead involved in exchanging messages is usually negligible.

**2.10 PERFORMANCE**

If a distributed system is to be used its performance must be at least as good as a centralized system. That is, when a particular application is run on a distributed system, its overall performance should be better than or at least equal to that of running the same application on a single processor system. However, to achieve his goal, it is important that the various components of the operating system of a distributed system be designed properly; otherwise, the overall performance of the distributed system may turn out to be

worse than a centralized system. Some design principles considered useful for better performance are as follows :

1. Batch if possible, Batching often helps in improving performance greatly. For example, transfer of data across the network in large chunks rather than as individual pages is much more efficient. Similarly, piggybacking of acknowledgement of previous messages with the next message during a series of messages exchanged between two communicating entities also improves performance.
2. Cache whenever possible : Caching of data at clients’ sites frequently improves overall system performance because it makes data available wherever it is being currently used, thus saving a large amount of computing time and network bandwidth. In addition, caching reduces contention on centralized resources.
3. Minimize copying of data : Data copying overhead (e.g. moving data in and out of buffers) involves a substantial CPU cost of many operations. For example, while being transferred from its sender to its receiver, a message data may take the following path on the sending side :
   1. From sender’s stack to its message buffer
   2. From the message buffer in the sender’s address space to the message buffer in the kernel’s address space
   3. Finally, from the kernel to the network interface board

On the receiving side, the data probably takes a similar path in the reverse direction. Therefore, in this case, a total of six copy operations are involved in the message transfer operation. Similarly, in several systems, the data copying overhead is also large for read and write operations on block I/O devices. Therefore, for better performance, it is desirable to avoid copying of data, although this is not always simple to achieve. Making optimal use of memory management often helps in eliminating much data movement between the kernel, block I/O devices, clients, and servers.

1. Minimize network traffic : System performance may also be improved by reducing internode communication costs. For example, accesses to remote resources require communication, possibly through intermediate nodes. Therefore, migrating a process closer to the resources it is using most heavily may be helpful in reducing network traffic in the system if the decreased cost of accessing its favorite resource offsets the possible increased post of accessing its less favored ones. Another way to reduce network traffic is to use the process migration facility to cluster two or more processes that frequently communicate with each other on the same node of the system. Avoiding the collection of global state information for making some decision also helps in reducing network traffic.
2. Take advantage of fine-grain parallelism for multiprocessing. Performance can also be improved by taking advantage of fine-giam parallelism for multiprocessing. For example, threads are often used for structuring server processes. Servers structured as a group of threads can operate efficiently, because they can simultaneously service requests from several clients. Fine-grained concurrency control of simultaneous accesses by multiple processes, to a shared resource is another example of application of this principle for better performance.

Throughout the book we will come across the use of these design principles in the design of the various distributed operating system components.

**2.11 SCALABILITY**

Scalability refers to the capability of a system to adapt to increased service load. It is inevitable that a distributed system will grow with time since it is very common to add new machines or an entire subnetwork to the system to take care of increased workload or organizational changes in a company. Therefore, a distributed operating system should be designed to easily cope with the growth of nodes and users in the system. That is, such growth should not cause serious disruption of service or significant loss of performance to users. Some guiding principles for designing scalable distributed systems are as follows :

1. **Avoid centralized entities :** In the design of a distributedoperating system, use of centralized entities such as a single central file server or a single database for the entire system makes the distributed system non-scalable due to the following reasons :

**Security :**

In order that the users can trust the system and rely on it, the various resources of a computer system must be protected against destruction and unauthorized access. Enforcing security in a distributed system is more difficult than in a centralized system because of the lack of a single point of control and the use of insecure networks for data communication. In a centralized system, all users are authenticated by the system at login time, and the system can easily check whether a user is authorized to perform the requested operation on an accessed resource. In a distributed system, however, since the client – server model is often used for requesting and providing services, when a client sends a request message to a server, the server must have some way of knowing who is the client. This is not so simple as it might appear because any client identification field in the message cannot be trusted. This is because an intruder (a person or program trying to obtain unauthorized access to system resources) may pretend to be an authorized client or may change the message contents during transmission. Therefore, as compared to a centralized system, enforcement of security in a distributed system has the following additional requirements :

**27**

1. It should be possible for the sender of a message to know that the message was received by the intended receiver.
2. It should be possible for the receiver of a message to know that the message was sent by the genuine sender.
3. It should be possible for both the sender and receiver of a message to be guaranteed that the contents of the message were not changed while it was in transfer.

**Cryptography** is the only known practical method for dealing with these security aspects of a distributed system. In this method comprehension of private information is prevented by encrypting the information, which can then be decrypted only by authorized users.

Another guiding principle for security is that a system whose security depends on the integrity of the fewest possible entities is more likely to remain secure as it grows. For example, it is much simpler to ensure security based on the integrity of the much smaller number of servers rather than trusting thousands of clients. In this case, it is sufficient to only ensure the physical security of these servers and the software they run.

**Revision Exercise:**

1. Explain the various transparencies of a distributed system
2. How are location, relocation and migration transparencies different from each other. Explain with examples.
3. Explain the flexibility of a DS.
4. Discuss the security aspects of a DS.

**REFERENCES:**

1) [*Dolev, Shlomi*](https://en.wikipedia.org/wiki/Shlomi_Dolev) (2000), Self-Stabilization, [*MIT Press*](https://en.wikipedia.org/wiki/MIT_Press), [*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*0-262-04178-2*](https://en.wikipedia.org/wiki/Special:BookSources/0-262-04178-2)*.*

2) Elmasri, Ramez; *[Navathe, Shamkant B.](https://en.wikipedia.org/wiki/Shamkant_Navathe" \o "Shamkant Navathe)* (2000), Fundamentals of Database Systems (3rd ed.), [*Addison–Wesley*](https://en.wikipedia.org/wiki/Addison%E2%80%93Wesley), [*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*0-201-54263-3*](https://en.wikipedia.org/wiki/Special:BookSources/0-201-54263-3)*.*