Distributed Systems: Concepts, Principles, Models and Algorithms

Article *in* Journal of Early Modern Studies · June 2022

CITATIONS
0

READS
78

5 authors, including:

Khalid K. A. Abdullah
King Abdulaziz University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Abdulaziz Alzubaidi
King Abdulaziz University
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Distributed Systems: Concepts, Principles, Models and Algorithms

Abdulaziz A. Al-Zubaidi, Khalid K.A. Abdullah, Muhammed K. Dauda
Mohammed S. Al-Yahya, Mohammed J. Al-Haddad
Faculty of Computing and Information Technology
King Abdulaziz University
Jeddah, Saudi Arabia

Abstract. Distributed systems properties, implementation, models, and architecture is a point of dilemma to many newbies, and also those with little knowledge, several explanations from non-practitioners are misleading. In this paper we discuss these concepts and clear the air for newbies to the field to have essential knowledge that can help them further their knowledge. To effetely give justice to the matter we must define several properties and models. We believed that all available systems satisfy these properties. The objective of this paper is to provide a roadmap to the reader with a better view of distributed system aspect, related implementation, and models.

Keywords: Distributed systems, Parallel computing, Shared memory, Message passing, Clock synchronization

Introduction

Distributed system is a collection of autonomous systems, workstations, and servers connected to provide services to client or user virtually using a protocol called middle ware which affairs as single unit to the user. The aim of distributed system is to share resources, increase availability, throughput, efficiency and overcome failure (overcome single point of failure) without end user knowing the underlying complex structure behind it.

Although applications of distributed systems are found in many organizations and in many dimensions ranging from network, software, and the intended use. Moreover, the phrase “distributed systems” runs in many mouths but it’s true implementation, concept, and features lack adequate understanding, new researcher find it difficult to concisely understand its terms, properties, and how to build-on on the current research and contribute their quotes.

Distributed system is sharing of resource (memory, processor, and data) for efficiency, overcome single point of failure, and provide availability.

Distributed Systems

A. Goals of distributed system

- Openness of the system: this term refers to the ability or degree in which new node, system or service can be added to the available program or service.
- Security: the openness and transparency of the system rises the issue of security, since service or call from one end machine can seamlessly interact and appears as one to end user, then the security of resource can be compromise.
- Scalability: system is scalable if the increase in demand of it resources from higher number of users and withstand or tolerate it without failure or slagging.
- Fault Tolerance: one of the factors of transparency in which one failure will not affect others, and the system can withstand and resist the failure and work as if nothing has happened preventing fault is directly proportional to increasing availability.
- Concurrency: shared resource is access simultaneously from different users, this capability most be possessed by all shared resources in a distributed system and the property is called concurrency.

B. Types of distributed system

1) The Architectural model:

- Client server: it the most implemented architectural design also called Master-slave, where the idea is like supplier and consumer, one part always requests for data, information, or resource and the other part provide the such. Although, every serve can also be a client to another server but cannot be both client and server at same time. And also servers must know about clients while client doesn't need to know the identity of servers.
- Peer-to-Peer: This Implementation is in contrast with the Master-Slave. Every system is a master on its own, data is distributed across each node but supper from duplicate information which might cost a lot for maintenance

2) The Fundamental Model:

- Interaction Model: in distributed system there is no provision of service unless those autonomous system interacts and coordinate within them self. Many processes are passed to within distributed system making it complex of interaction. Therefore, the way, manner, and state this system taking part in the interaction produces distributed algorithms to measure performance and take care of global time or sequence.
- Failure Model: since network is backbone of the distributed system, and network is not reliable, failure might occur at any point. Therefore, there must be a way to understand and accept failure if it occurs. There are several types of failures, timing failure, arbitrary failure, and others like omission failure.
- Security model: security of distributed system is highly essential to the successful transaction of processes and channels to protect the object, the task and application itself.

3) The Physical Models:

- A distributed computing system is a group of separate computers that seem to consumers as a single computing system. The computers in a distributed system can be near each other and connected via a local area network, or they can be geographically separate servers and data stores that exist in different systems around the world. These components may interact, communicate, and work together to achieve the same goal, creating the appearance of a single, unified system with significant processing capabilities. It has two subgroups system. Cluster computing and Grid computing.
- Distributed Pervasive: Pervasive computing refers to the presence of embedded microprocessors in the objects around us that people are completely unaware or unmind of. It also known as Internet of Things; it aims to upgrade the living standard in our daily lives while reducing man-machine interaction.
- Distributed information systems: this type focuses on the concept of data sharing and access to data in appropriate ways. Distributed information systems has a major role in facilitating the expansion of the organization on large scales, this refers due to the lack of reliance on the centralization of data.

Parallel Computing

In the past, the devices were working on a single-processor structure, so that the problem was taken completely and divided into a set of instructions, then each instruction was solved and the final solution was provided for all instructions, then science advanced and technology developed and the term synchronization appeared, and it is similar to the previous model in terms of having one processor, but it became the processor inputs for each instruction or task a portion of time and then takes the next instruction or task. The user feels that the processor can process a set of commands at the same time, but in reality, the

processor has a great ability to solve a set of tasks separately and simultaneously, after this development appeared a new term is parallel computing.

Parallel computing is the process by which several task can be done hand in hand with a several processors (task handlers) without interrupting one another or waiting for any other. It is the opposite of sequential processing where tasks are processed by a single processor (task handler) or one must wait for another to finish. In parallel computing the idea of sequential processing is broken consequently increase efficiency and performance.

There are many definitions about parallel computing. One of the definitions is that parallel computing represents the execution of tasks by many processors at the same time and that the processors may connected to shared memory, or the memory may be separate. Another definition that the "parallel computing is the simultaneous use of multiple compute resources to solve a computational problem".

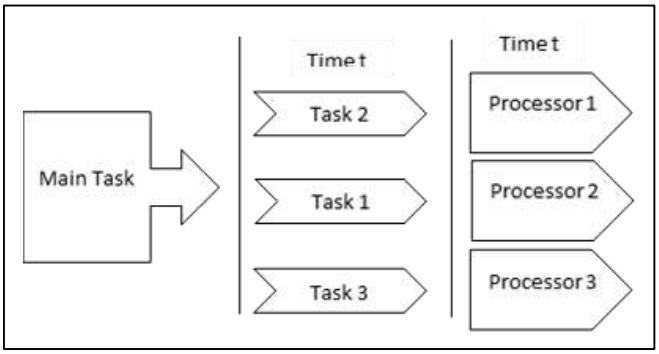

Figure 1. Parallel computing process

From the previous definitions, it is clear that parallel computing depends on working across several processors, which means that one problem is divided and solved across many processors at the same time instead of waiting and solving all the problem through one processor as it was in the past, this concept is very broad as it depends on the division the problem is based on a set of tasks, and each processor takes one task in parallel with the other processors, in order to provide the final solution faster.

A. Flynn's Taxonomy

There are many definitions and classifications based on how to build and operate computers and applications. The purpose of these classifications is for academic purposes as well as for software and hardware development. The most famous of these classifications is the so-called Flynn Classification. Flynn divided parallel computing into four divisions, namely Single Instruction Single Data SISD, Single Instruction Multiple Data SIMD, Multiple Instruction Single Data MISD, and Multiple Instruction Multiple Data MIMD.

The divisions depend on two main axes, which are data and instructions. These two axes are either single or multiple. Before going into the details of Flynn's classifications, we explain the architecture. Parallel computing architecture is based on parallel bits, parallel instructions, and parallel tasks.

- Parallel bit: Modern devices take a set of data (bits) at the same time, for example, in the old devices there were 8-bit computers, then developed and computers appeared with stronger processors and larger data lanes (bus) and 16-bit processors appeared and then developed after That devices and became a 32-bit processing capacity and also after that evolved and became a 64-bit processing capacity and so on until the super devices appeared, this can be summarized by saying that the ability of a computer to transmit data at the same time is called parallel (bit) data.

- Parallel Instructions: In the past, the processor could take one instruction at a time until the processors evolved. Multiple or parallel processors appeared, which is the ability of the processor to take more than one instruction at the same time, and this undoubtedly speeds up the process of solving the problem significantly if the application or program supported the possibility of multi-tasking.
- Parallel tasks: The concept of parallel tasks is the ability of applications to run several tasks in parallel.

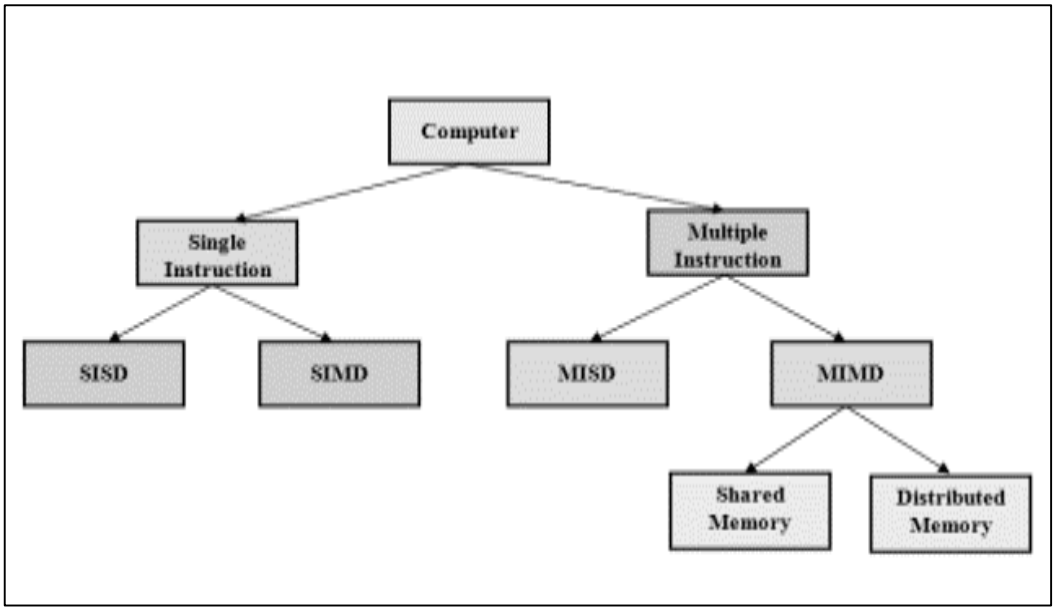

Figure 2. Classification of computer architectures

1) SISD

The Single Instruction Single Data is the first type of Flynn partition, and this type is a single-core computer with one processor that is not parallel, and the processor in one circuit receives one instruction and receives one data packet, and the process is repeated until the completion of the complete problem solution, this type is one of the oldest types of devices.

2) SIMD

The Single Instruction Multiple Data is the second type of Flynn's classification, and this type takes data in parallel, but it executes a single instruction for all data, which means that it is not possible to take a set of data and execute different commands on it, this type is often used in image and audio applications that use an instruction One for a large set of data, for example, when we want to lighten an image or make it larger or smaller, that we take the image and execute the desired command on each pixel at the same time. Solo Another example is when we amplify the volume, we increase the power rating of all digital data.

3) MISD

The multiple instruction for a single data is the third type of Flynn's classification. This type is based on the work of a set of different instructions through a group of processing units in parallel on a single data packet. This type is used in applications that work a set of commands on single data, such as cipher-breaking programs, as they Execute a set of commands on the same data packet.

4) MIMD

Multiple instructions for multiple data is the fourth division of Feline divisions and this type depends on executing a set of commands in parallel and through a group of processing units on a set of different data at the same time, this type of computer structure is often used in super devices because of its high capacity.

One of the most prominent negatives of this type of structure is that it needs a large processing capacity, and therefore a high cost, and one of its most prominent advantages is the great ability to analyze and is often used in the analysis of big data.

The multiple instruction and multiple data model and the single instruction and multiple data model use two main structures in data usage which are the concept of shared memory and the concept of distributed memory.

In distributed memory, each processor has its own memory in which it is directly connected and fetching data is fast and there is no intersection with other processors other than the concept of shared memory.

B. Shared Memory

Shared memory is memory that has access from a group of parallel processors and each processor has an update about the memory of the same other processors. The data in this model is often transferred via the so-called bus, one of the advantages of this system is the high speed for direct access to memory without the so-called message passing and the next thing this model easier for programmer, the most prominent negative of this model is the conflict of memory access request.

C. Distributed Memory

Distributed memory is a structure in which each processor has a special memory with which it communicates directly, in this structure (distributed memory) there are no limits to expansion, not even to productivity because each processor has its own memory. In this model, a new concept appears, which is "Message Passing".

Distributed Systems Communication Paradigm

Processes frequently need to communicate with one another, and the communication processes follows various workstations, this coordination makes distributed systems difficult. Inter-process communication allows processes to collaborate and shared variables that can be synchronized.

Message in a simple term is the processes, way and manner nodes in a distributed system communicate and coordinate their activities through network. These messages can be one of the following types: Message Passing, Remote Procedure Call, and Remote Method Invocation

The most frequent technique of inter-process communication in distributed systems is message Passing and remote procedure calls. Tasks communicate with one another by sending and receiving simple messages. Data transmission often necessitates cooperative procedures done by each component.

Message passing is the mostly implemented inter-process communication in distributed systems. It is the one that requires a programmer to identify the source, destination, the call (message) and it type. Message passing can also be seen as an exchange of messages across processes by sending and receiving messages using basic primitives. It is essential for the programmer to understand the message as well as the names of the source and destination processes that is the main reason parallel computing processes utilize the advantage of massage passing interface (MPI) standard to implicitly defined and designed according to programmer's need.

Data transmission often necessitates cooperative procedures done by each component. Most of the inter-process communication in distributed systems is based on message forwarding. It is the most basic level of abstraction and needs the application programmer to understand the destination process, the message, the source process, and the data types anticipated from these processes. Message passing can be defined as an exchange of

messages across processes by sending and receiving messages using basic primitives. It is essential for the programmer to understand the message as well as the names of the source and destination processes. Remote procedure calls have a higher level of abstraction than message passing; remote procedures are called in the same way that local procedures are, with the operating system handling the details of locating the remote procedure and preparing the arguments for inclusion in the message that calls the remote procedure (Bova et al., 2001).

The syntax of the message contain send (R, M) and received (S, M) where R is the receiver, S, Sender and M as Message. However, the type these messages can be of two types: Synchronous Blocking, and Asynchronous Unblocking.

A. Remote procedure call (RPC)

Unlike message passing, where programmer explicitly define message and control movement across. It has higher level of abstraction than message passing, it's called in the same way the local procedure is call, with the operating system handling the details of locating the remote procedure and preparing the arguments for inclusion in the message that calls the remote procedure (Hamilton, 1984). The procedures are already predefined with the argument and its return values. The node (remote caller) calls the procedure, the server replies, and caller received feedback as follows:

- Call: Remote_Procedure_Name (V, R).
- Receiver: Remote_Procedure_Name (Incoming_V, Output_R).
- Reply: (C, R) where V is the value argument, R is result parameter and C is the initial caller Id.

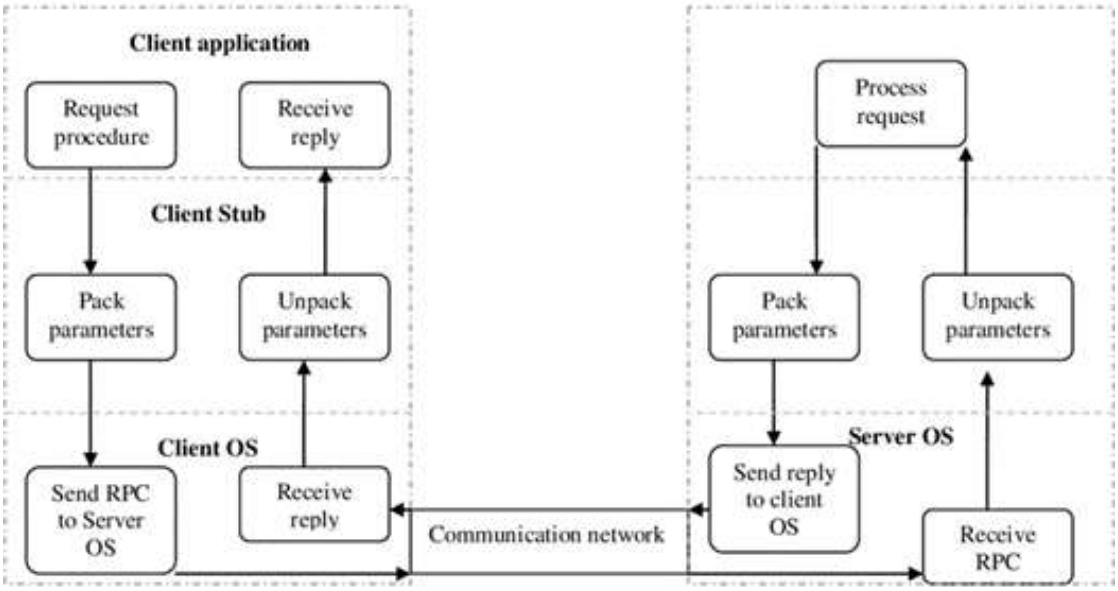

Figure 3. Remote procedure call architecture

RPC Methodology

- The client process makes a parameterized call to the client stub, and its execution is halted until the call is finished.
- The arguments are then marshalled through client stubs and converted into machine-independent form. The message is then prepared, which includes a representation of the parameters.
- To determine the identification of the site, the client stub communicates with the name server where the remote process is located.

- The client stub transmits the message to the site where the remote procedure call exists using the blocking protocol. This step stops the client stub until it receives a response.
- The message delivered from the client is received by the server site and converted into machine-specific format.
- The server stub now makes a call to the server procedure with the arguments, and the server stub is terminated until the procedure is completed.
- The server method passes the generated results to the server stub, where they are translated into machine-independent format and a message containing the results is created.
- The result message is forwarded to the client stub, where it is transformed back into machine-specific format that the client stub can understand.
- Stub provides the results to the client process in the end (Tay & Ananda, 1990).

B. Remote Method Invocation (RMI)

Remote Method Invocation (RMI) is a technique that allows Java objects to be shared across Java Virtual Machines (JVM) via a network. An RMI application is made up of a server that generates remote objects that conform to a defined interface and are accessible for method invocation to client programs that get a remote reference to the object. When a remote item is transmitted from one virtual machine to another, RMI considers it differently from a local object. RMI sends a remote stub for a remote object rather than a copy of the implementation object to the receiving virtual machine. The stub functions as the distant object's local equivalent, or proxy, and is essentially the remote reference to the caller. The caller calls a method on a local stub that is responsible for executing the call on remote objects, and the remote object stub develops and implements the same set of remote interfaces as the remote object This enables a stub to be cast to any of the interfaces implemented by the remote object. This, however, implies that only the methods described in a remote interface are available for use in the receiving virtual machine (Maassen et al., 2001).

RMI has the unusual ability to dynamically load classes from one JVM to the other through their byte codes, even if the class is not declared on the receiver's JVM. This implies that new object types may be introduced to an application just by changing the classes on the server, with no further effort required on the receiver's side. This transparent loading of new classes through their byte codes is a unique RMI feature that substantially facilitates program modification and update.

RMI Methodology

Creating a remote interface is the first stage in developing an RMI application. A remote interface is a subclass of (java.rmi.Remote), indicating that it is a remote object with methods that may be called across virtual machines. Any object that implements this interface is referred to as a remote object. An interface representing an object that can be serialized and transferred from JVM to JVM must also be established to demonstrate dynamic class loading in action. The interface is a subclass of the interface (java.io.Serializable). RMI transports objects by value across Java virtual machines via the object serialization technique. Implementing Serializable indicates that the class is capable of being converted into a self-describing byte stream, which can be used to rebuild an exact replica of the serialized object when read back from the stream. Any entity of any type can be given to or from a remote method if it is an instance of a basic data type, a remote object, or an object that implements the interface (java.io.Serializable). Remote items are passed mostly via reference. It acts as a client side agent, the so-called reference of the remote object (stub), and this agent develops

the set of remote interfaces that are executed on the remote object. Object serialization is used to send local objects by copy. By default, all fields are copied, with the exception of those tagged static or transitory. On a class-by-class basis, the default serialization behavior can be modified.

Thus, distributed application clients can dynamically load objects that implement the remote interface even if they are not specified in the local virtual machine. The next step is to implement the remote interface, which must include defining a constructor for the remote object as well as all of the methods stated in the interface. The server must be able to create and install remote objects after the class is built. For bootstrapping purposes, the server is initialized by establishing and installing a security manager, creating one or more instances of a remote object, and registering at least one of the remote objects with the RMI remote object registry (or another name service like as one that utilizes JNDI). An RMI client works in the same way as a server; after installing a security manager, the client creates a name that can be used to search for a distant object. The client looks up the remote object by name in the remote host's registry using the (Naming.lookup) function. The code generates a URL that indicates the host where the server is executing when performing a name lookup.

C. CORBA

The Object Management Group (OMG) has clearly defined Common Object Request Broker Architecture (CORBA) and describes it as standards that allow communication between software components that are programmed in different programming languages and run under different operating systems as well. CORBA serves as a standard for communication across distributed networks that allows objects to be distributed within networks so that commands and actions on those objects can be executed remotely. CORBA, by its nature, is not specifically related to any programming language, but any language related to CORBA can require and develop CORBA objects. Objects are specified using an interface definition language syntax (IDL) (Henning & Vinoski, 1999).

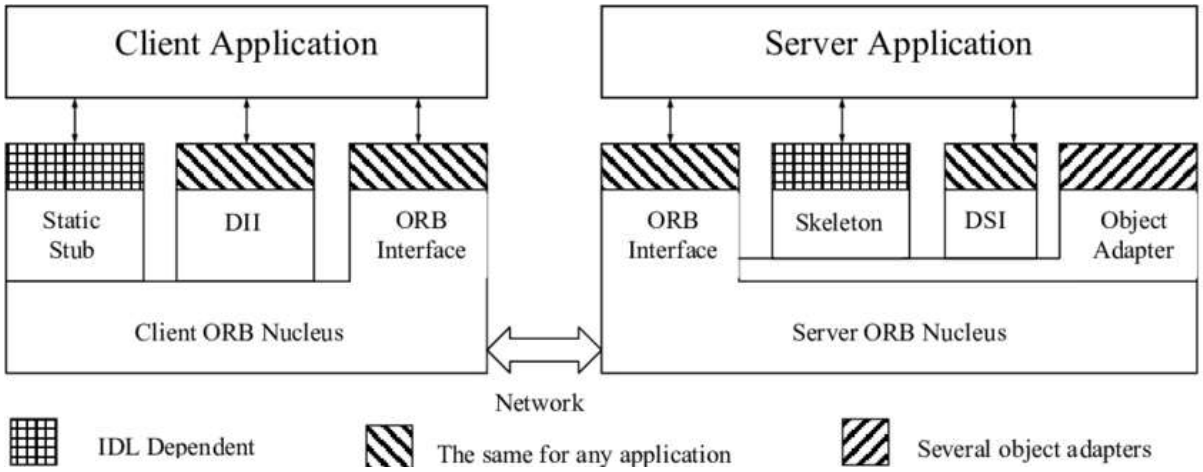

Figure 4. Common Object Request Broker Architecture (CORBA)

CORBA consists of four components:

- Object Request Broker (ORB): An ORB manages parameter communication, marshaling, and un-marshaling such that parameter handling is transparent to CORBA server and client programs.
- CORBA server: The CORBA server is responsible for creating CORBA objects and initializing them using an ORB. The server stores references to CORBA objects in a naming service, allowing clients to access them.

- Naming service: The naming service stores CORBA object references.
- CORBA Request node: This node serves as a CORBA client (Schmidt & Kuhns, 2000).

CORBA Methodology

The first step in developing a CORBA application is to create the remote object's interface using the OMG's interface specification language (IDL). When the IDL file is compiled, two types of stub files are produced: one that implements the client side of the program and another that implements the server. Clients and servers are linked and interconnected in the form of stubs and skeletons because IDL defines the interfaces with high accuracy, so the server skeleton does not interact with the client stub even if the heel and skeleton are written in different programming languages and run on different operating systems and use separate ORBs.

The client then retrieves its object reference via the Orb before invoking the distant object instance. The client uses the same code as it would for a local invocation, but instead of an instance of a local object, it uses an object reference to the remote object. When the ORB evaluates the object reference and determines that the target object is remote, it marshals the parameters and sends the invocation over the network to the remote object's ORB rather than another process on the same computer (Keahey & Gannon, 1997).

CORBA also allows for the dynamic discovery of information about distant objects during execution. Each interface has a different method. First, the type information is generated by the IDL adapter, and then that information is stored in the IR interface repository file, and then the client retrieves and queries the IR to get the runtime information about a specific interface, and the Then I will use that knowledge dynamically.

DII is used for some tasks such as building and calling a method on a CORBA server object, as well as another task such as calling the Dynamic Skeleton Interface (DSI) which in turn allows the client to perform some remote tasks on the server side without knowing the type of object being implemented.

CORBA is sometimes seen as a shallow specification since it is concerned with syntax rather than semantics. CORBA provides a vast number of services that can be supplied, but only to the extent of identifying which interfaces application developers should utilize. Unfortunately, outside of the fundamental capabilities offered, the very minimum that CORBA expects from service providers makes no mention of security, high availability, failure recovery, or guaranteed behavior of objects, and CORBA considers these characteristics optional. As a result of the lowest common denominator strategy, ORBs vary so widely from vendor to vendor that it is exceedingly difficult to build portable CORBA code since critical capabilities such as transactional support and error recovery are inconsistent between ORBs. Fortunately, with the creation of the CORBA Component Model, which is a superset of Enterprise Java Beans, much of this has changed.

Timing and Concurrency Control

Distributed Systems use message passing as a means of communication between nodes. Transactions and events must be executed sequentially in many real-time applications executed on distributed systems, such as banking systems and reservation systems. Organizing events in a logical sequence is critical to maximizing the use of resources available and minimizing waste.

A. Concurrency Control in Distributed Systems and Synchronization

Concurrency: The Idea of concurrency is the same as multitasking, which is performing many tasks by a single processor in a shared memory. A single shared memory (resource) is accessed by single processor to complete several pieces of tasks to increase throughput. It's also differ with sequential processing. The fig below shows the Illustrated idea of

concurrency. The processor switch between tasks with or without explicitly finishing or completing it, the switching between the several tasks is so fast that user thinks it done in parallel way.

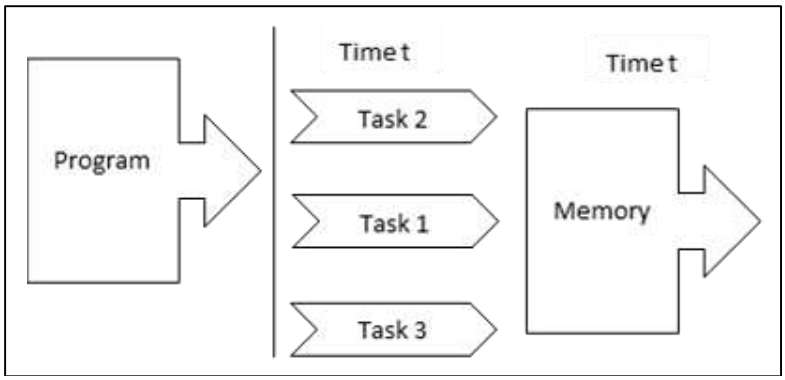

Figure 5. Concurrent process

B. Time in Distributed Systems

Time is a crucial and interesting issue in distributed systems; we frequently want to estimate time correctly for various reasons. It is essential to synchronize a computer's clock with an approved, external time source to determine what time of day some event occurred. For example, a merchant's computer and a bank's computer are both involved in an e-commerce transaction. It's important for auditing purposes that those occurrences are appropriately timestamped. Using clock synchronization, this can be done. An assigning of available resources and coordination between the various processes that make up a project is necessary. Time synchronization is required to resolve conflicts. Multiple types of clock synchronization are used to keep distributed systems in sync (Lamport, 1978).

C. Physical Clocks Synchronization

Today, most computers use CMOS clock circuits powered by quartz resonators to maintain track of time. So, even if the system is not running, the clocks continue to tick. A clock circuit is a Programmable Interval Timer in older Intel architectures; an (Advanced Programmable Interrupt Controller), or APIC, in newer systems, is often programmed by an operating system to emit an interrupt periodically when it is enabled. Standard times are 60 or 100 times for a second. The interrupt handler procedure only increments the value of a memory counter by one each time it is invoked. Because they are sensitive to temperature and acceleration changes, the highest quartz resonators can attain an accuracy of one second in 10 years. However, as they age, their resonating frequency might shift, making it difficult to measure time accurately. When used at 31°C, standard resonators have an accuracy of 6 parts per million, which equates to 12 seconds of accuracy each day (Khandare et al., 2013).

1) Coordinated Universal Time (UTC):

A worldwide timekeeping standard known as UTC can be used to synchronize computer clocks with outside sources of high accuracy. "Leap seconds" are added or taken away to keep it in sync with the astronomical time. It depends on atomic time. In addition to land-based broadcast stations and satellites, UTC signals are broadcast from a variety of locations around the world. Time signals are published on numerous shortwave frequencies, such as WWV in the United States. "The Global Positioning System" (GPS) is one example of a satellite source (GPS). Apart from the 1013 drift rate is used in the better accurate physical clocks. These clocks use atomic oscillators. The output of these (atomic clocks) is utilized as the standard for expired real-time, known as International-Atomic-Time. The transition between Caesium-133's two hyperfine ground states has been estimated as "9,192,631,770" periods since 1967 (Cs133). Time units such as seconds, years, and others

are derived from astronomical time. At the time of their original definition, they were expressed in terms of Earth's axial rotation and solar rotation. As a result of tidal friction, the Earth's rotation around its axis is becoming longer over time, with interim increases and decreases due to atmospheric impacts and convection currents within the Earth's core. As a result, the ephemerides between atomic and astronomical time are frequently seen (Tadayon, 2019).

2) Cristian's algorithm for synchronizing clocks:

Cristian, also known as CRI, presented the idea of a probabilistic clock synchronization algorithm. To synchronize computers externally, Cristian (1989) suggested using a time server linked to a machine that carries data from a source of UTC (Bova et al., 2001). Cristian's algorithm depends on a remote clock reading ("RCR"). A node uses RCR to read a remote node's clock with a defined minimum precision. RCR entails requesting the time on a target node's clock. The requesting node then uses the response to approximate the destination node's clock time. The RCR expresses the largest estimation error (D - dmin), where D is 1/2 the response time, and 'dmin' is the shortest period. CRI is a client-server algorithm that achieves synchronization using RCR. The time server receives a message from each client requesting the current time. Each client resynchronizes with the server regularly by utilizing RCR to estimate the readings on the time server's clock and modify its own clock accordingly. Resynchronization is not ensured, however, because RCR may be unable to communicate or understand. Analytically, the likelihood that algorithm CRI will fail to resynchronize can be calculated. As a result, CRI is a 'probabilistic' clock synchronization algorithm that is not bound by the set limit. As a result, compared to deterministic algorithms, CRI can ensure substantially reduced maximum clock skews.

3) Berkeley's algorithm for synchronizing clocks:

Gusella and Zatti present an internal synchronization algorithm they created for a cluster of Berkeley UNIX machines. It selects a coordinator computer to serve as the master. Unlike Cristian's protocol, this computer polls the slaves, or computers whose clocks must be synced regularly. It receives the clock values from the slaves. The master calculates their (local clock times) via monitoring round-trip times (a technique similar to Cristian's) and averaging the results (including its own clock's reading). Individual clock inclinations to move fast or sluggish should be canceled out by the average time. It sends each node the offsets by which its clock has to be adjusted rather than sending the modified clock time back to the slaves, which would create further confusion due to network delays, for example. Three machines had (3:00, 3:25, and 3:50) as their times. The server is the main node with a time of 3:00. (master). It sends a synchronization request to the rest of the group's machines. In response to the query, each of these devices delivers a timestamp. The server now computes ((3:00+3:25+3:50)) /3 = 3:05 by averaging the three timestamps it received and its own. It currently transmits offsets to each machine, which synchronizes the machine's time to the average after the balance is implemented. The device with a time clock of 3:25 receives a -0:20 offset, whereas the device with a timepiece of 3:50 receives a +0:15 offset. The server's time must be adjusted by +0:05. The algorithm also contains a feature that ignores readings from clocks with a large skew. The master can compute a fault-tolerant average, which averages values from devices with clocks that haven't drifted more than a specified amount. If the master nodes fail, any slave can be chosen to take its place.

D. Logical Clock Synchronization

Consider the issue of assigning sequence numbers ("timestamps") to events that may be agreed upon by all participating processes. What counts in these circumstances is that all procedures can concur on the ordering in which linked events occur, not really the time of day the event occurred. Our goal is to obtain system-wide event sequence numbers. These

timepieces are known as logical clocks. If we can do this for all the occurrences in the system, we achieve total ordering: each event is given a special timestamp (number), and each of these timestamps is unique. However, total ordering isn't always required. We don't care when processes interact because we don't matter when their events happen. Partial ordering occurs when we are only interested in assigning timestamps to linked (causal) occurrences. In a distributed system, a logical clock is a technique for capturing temporal and causal links.

1) Lamport's Clock algorithm:

Lamport (1978) devised a logical clock for synchronization, which is a simple algorithm for capturing what happened before ordering numerically. A "Lamport logical clock" is a software counter with a monotonically growing value that does not have to be related to any (physical clock). To represent the link between events, Leslie Lamport devised a "happens-before" notation: ab denotes that a occurs before b. If A is the sent time & B is the received date, then AB should be valid; content cannot be obtained before it is sent. This is a transitive relationship. If AB and BC are true, then AC is true. If (A and B) are happenings in the same process, then AB is true if a happens before B. The relevance of logical time measurement lies in giving a timestamp to each occurrence so that everyone can agree on the ultimate sequence of events. Because the clock ((our timestamp generator)) cannot go backward, if AB, then clock(A) clock(B). If A and B occur on separate processes that do not communicate (even through third parties), then AB is false. Those occurrences are said to be simultaneous since A could not have influenced B.

2) Mutual Excision Clock Synchronization

The activities of distributed processes frequently need to be coordinated. When a group of operations shares a source or set of resources, mutual exclusion is frequently used to avoid interference and assure consistency in resource access. This is an important sector problem that is well-known in the operating system area. In general, shared variables and capabilities provided by a single regional kernel cannot be leveraged to solve the problem in a distributed system (Ganguly & Lemmon, 1999). We need a distributed mutual exclusion solution that is only dependent on message passing. Servers that manage shared resources and provide methods for mutual exclusion are used in some circumstances. Consider the case of users who make changes to a text file. Allowing them to access the file just one at a time and forcing the editors to freeze the file before modifications can be made is a simple way to ensure that their edits are consistent (Coulouris, 2012).

3) Lamport's Mutual Exclusion Clock Algorithm

Lamport was the first to propose a distributed mutual exclusion algorithm based on his logical clock notion. Each process is considered to have a unique identifier, which is presumed to be a positive integer for convenience. Single words of memory are allowed atomic reads and write, which are presumed to be large enough to carry a process's number. It is expected that the critical part and all code beyond the mutual exclusion mechanism do not change any of the algorithms' variables. This algorithm is not reliant on tokens. Queries for Critical Section are ordered using timestamps in non-token-based protocols. The message of request Includes the following: Resource name, Identifier, (device id, process id) the time stamps. When sites issue Critical Section requests, the operating system assigns them a timestamp, which is a unique integer value. When an order is received, the timestamp is incrementally incremented. Requests for smaller timestamps take precedence over requests for larger timestamps. The request set Ri=P1, P2, P3,..... Pn for a site Pi in Lamport's scheme. It contains all of the locations from which Pi must obtain permission before proceeding to the Critical Section. In increasing the sequence of timestamps, each process keeps a queue of pending requests to enter the critical section (Yadav, Yadav, & Mandiratta, 2015).

4) Centralized Algorithm for Mutual Exclusion

One coordinator oversees all requests for access to the shared resource/data, as the name implies. Every process asks the coordinator for permission, and only if the coordinator authorizes it is a process allowed to enter Critical Section. In a queue, the coordinator will maintain track of the requests. Process 1 requests authorization from the coordinator to access a crucial region. Because the line is empty and there are no pending requests, permission is granted by the coordinator. The second process then requests permission to visit the same critical zone as the first. Because Process one has not yet exited CS, the controller does not respond. When Process 1 leaves the critical section, it informs the coordinator, who then grants permission to Process 2. The most straightforward mechanism to establish mutual exclusion is by using a server that gives access to the vital region. A process enters a critical area by sending a query packet to the server and then waiting for a response. The response is a token granting access to the vital part in terms of concept. When no process possesses the (token) at the moment of the query, the server gives the token immediately. If another method already has the pass, the server will not respond and will instead queue the request. When an operation completes the critical part, it sends a request to the server and returns the token to it.

5) Distributed Algorithm for Mutual Exclusion

There are no coordinators in this algorithm. Each process requests permission to access the Critical Section from the other functions. Two subtypes are distinguished: Containment-based algorithms and Controlled ("Token-based") algorithmic techniques. The algorithm's key characteristic is the usage of well-known decentralized algorithm methods simultaneously. The knowledge-transfer control mechanism has been included to limit the number of packets required by the message-routing protocol. Sites can communicate with a group of each other. Distributed algorithms can also be separated down into timestamp-based and voting schemes (Yadav, Yadav, & Mandiratta, 2015).

6) Token Based Algorithm for Mutual Exclusion

Using token-based algorithms, all processes in the system have access to the same (privilege message token). The privilege message is required for a process to enter the Critical Section. For access to CS, a process must first send a request to a group of other processes known as its call set and then wait for the token to arrive. Sequence numbers are commonly used to distinguish between historical and current requests made by the same procedure in token-based algorithms. If an algorithm uses tokens, it generates less message traffic than one that does not. Algorithms based on tokens are immune to a deadlock since each token has a unique identifier. Because of this, their failure-to-failure tolerance is low, as it requires an extensive and time-consuming process to regenerate and recover tokens in the event of a token failure or loss. Mutual exclusion algorithms based on tokens fall into the following categories: Algorithms that use broadcasting and logical structures (iii) Token-based algorithms that are centrally controlled. Algorithms Using Broadcast Data Sites in the request set simultaneously send out request messages to all sites and then waits for the response. This is known as a "broadcast-based algorithm." The site must first receive the privilege notification for entering CS (Swaroop & Singh, 2007).

Conclusion

Distributed system is a broad subject that need researcher concern, for quality issue, failure resilience, efficient and optimized algorithm, because it comprises of many aspect of database, network engineering, software engineering and algorithms. Several efforts have been place but still there exist a gap for researcher to explore and contribute to the field.

The most similarities between distributed system and parallel system is the use of network interface that serve as a communication channel between nodes. And the most similarities between concurrency and parallelism is that both terms refer to the execution of multi-tasks in the same time frame. But concurrency does not necessarily imply that those tasks are simultaneously running at any given time. Instead, concurrency can be achieved on a single processor by the concept of multiprogramming while parallelism cannot be done on a single processor. It is critical to maximizing the use of resources available and minimizing waste.

References

Bova, S. W., Breshears, C. P., Gabb, H., Kuhn, B., Magro, B., Eigenmann, R., ... & Scott, H. (2001). Parallel programming with message passing and directives. *Computing in Science & Engineering, 3*(5), 22-37. doi: 10.1109/5992.947105.

Coulouris, J. D. G. (2012). *Distributed Systems: Concepts and Design*. DISTRIBUTED SYSTEMS (p. 1067).

Ganguly, J., & Lemmon, M. D. (1999). *Theory of clock synchronization and mutual exclusion in networked control systems*. Technical Report of the ISIS Group at the University of Notre Dame ISIS-99-007.

Hamilton, K. G. (1984). *A remote procedure call system* (p. 117). University of Cambridge PhD thesis.

Henning, M. & Vinoski, S. (1999). *Advanced CORBA programming with C++*. Reading, Mass: Addison-Wesley.

Keahey, K. & Gannon, D. (1997). PARDIS: A parallel approach to CORBA. In *Proceedings: The Sixth IEEE International Symposium on High Performance Distributed Computing* (Cat. No.97TB100183), Portland, OR, USA, 1997, pp. 31–39. doi: 10.1109/HPDC.1997.622360.

Khandare, N., Bhavsar, M. Kumare, P., & Raksha, S. (2013). Clock Synchronization in Distributed System. *International Journal of Computer Science and Information Technologies, 4*(3), 457-460.

Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM, 21*(7).

Maassen, J., Van Nieuwpoort, R., Veldema, R., Bal, H., Kielmann, T., Jacobs, C., & Hofman, R. (2001). Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems, 23*(6), 747-775.

Schmidt, D. G., & Kuhns, F. (2000). An overview of the real-time CORBA specification. *Computer, 33*(6), 56-63. doi: 10.1109/2.846319.

Swaroop, A., & Singh, A. K. (2007). A study of token-based algorithms for distributed mutual exclusion. *International Review on Computers and Software, 2*(4), 302-311.

Tadayon, T. (2019). *Time Synchronization in Distributed Systems without a Central Clock*. Master's thesis, University of Waterloo.

Tay, B. H. & Ananda, A. L. (1990). A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev., 24*(3), 68-79. doi: 10.1145/382244.382832.

Yadav, N., Yadav, S., & Mandiratta, S. (2015). A review of various mutual exclusion algorithms in distributed environment. *International Journal of Computer Applications, 129*(14), 11-16. doi: 10.5120/ijca2015907089.