A BRIEF UNDERSTANDING ON OBJECTIVES OF DISTRIBUTED SYSTEMS

Prof. K. Jagan Mohan
Vijayawada
Email: dr.kammili.jaganmohan@gmail.com

Abstract: It is very much necessary to know the important objectives of the distributed systems along with their design aspects in order to meet the emerging requirements of the present distributed systems for making future successful distributed systems which would serve the user requirements in terms of network connectivity, data transmission without any delay, security and confidentiality of information. For this, the goals and types of various distributed systems need to be studied and known properly along with how consistency and replication takes place in distributed systems– should be understood. This is all discussed very clear in this paper.

Keywords: Threads, Virtualization, Code migration, Remote Procedure Call, Synchronization, Consistency and Replication, Fault tolerance

Introduction:

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

A distributed system consists of autonomous components (communication devices i.e., computers, mobiles etc). Here, the users means that they may be people or programs. In one way or the other the autonomous components need to collaborate each other. How to establish this collaboration lies at the heart of developing distributed systems.

Distributed systems are often organized by means of a layer of software-that is, logically placed between a higher-level layer consisting of users and applications, and a layer underneath consisting of operating systems and basic communication facilities. Each application in the distributed system is offered the same interface. The distributed system provides the means for components of a single distributed application to communicate with each other, but also to let different applications communicate. At the same time, it hides, as best and reasonable as possible, the differences in hardware and operating systems from each application.

Objectives: The main goal of a distributed system is to make it easy for the users and applications to access remote resources, and to share them in a controlled and efficient way. Resources may include printers, computers, storage facilities, data, files, Web pages, and networks etc. As connectivity and sharing increase, security is becoming increasingly important. The second important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers. Access transparency deals with hiding differences in data representation and the way that resources can be accessed by users. Location transparency refers to the fact that users cannot tell where a resource is physically located in the system. Naming plays an important role in achieving location transparency. In particular, location transparency can be achieved by assigning only logical names to resources which gives no clue about the physical location of its main Web server. Distributed systems in which resources can be moved without affecting how those resources can be accessed are said to provide migration transparency. Sometimes resources can be relocated while they are being accessed without the notice of user or application; this

is referred to as relocation transparency. An example of relocation transparency is when mobile users can continue to use their wireless laptops while moving from place to place without ever being (temporarily) disconnected. Resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed. Replication transparency deals with hiding the fact that several copies of a resource exist. To hide replication from users, it is necessary that all replicas have the same name. Consequently, a system that supports replication transparency should generally support location transparency as well, because it would otherwise be impossible to refer to replicas at different locations. In cases, where one user does not notice that the other user is making use of the same resource from a shared database, it is referred to as database concurrency. An important issue is that concurrent access to a shared resource leaves that resource in a consistent state. Consistency can be achieved through locking mechanisms, by which users are, in turn, given exclusive access to the desired resource. This can be achieved through concurrency transparency in distributed systems. Making a distributed system failure transparent means that a user does not notice that a resource (he has possibly never heard of) fails to work properly, and that the system subsequently recovers from that failure. Masking failures is one of the hardest issues in distributed systems. Another important goal of distributed systems is openness. An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services. For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols. In distributed systems, services are generally specified through interfaces, which are often described in an Interface Definition Language (IDL). They specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised, and so on. Interoperability characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard. Portability characterizes to what extent an application developed for a distributed system **A** can be executed without modification, on a different distributed system **B** that implements the same interfaces as **A**. Another important goal for an open distributed system is that it should be easy to configure the system out of different components. Also, it should be easy to add new components or replace existing ones without affecting those components that stay in place. To achieve flexibility in open distributed systems, it is crucial that the system is organized as a collection of relatively small and easily replaceable or adaptable components.

The need for changing a distributed system is often caused by a component that does not provide the optimal policy for a specific user or application. As an example, consider caching in the World Wide Web. Browsers generally allow users to adapt their caching policy by specifying the size of the cache, and whether a cached document should always be checked for consistency, or perhaps only once per session. However, the user cannot influence other caching parameters, such as how long a document may remain in the cache, or which document should be removed when the cache fills up. Also, it is impossible to make caching decisions based on the *content* of a document. For instance, a user may want to

cache specific information he wants such as railroad timetables etc. Hence there should be a need for separation between a policy and mechanism. In the case of Web caching, for example, a browser should ideally provide facilities for only storing documents, and at the same time allow users to decide which documents are stored and for how long. In practice, this can be implemented by offering a rich set of parameters that the user can set (dynamically). Even better is that a user can implement his own policy in the form of a component that can be plugged into the browser.

Another goal of distributed systems is scalability. A system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system. Second, a geographically scalable system is one in which the users and resources may lie far apart. Third, a system can be administratively scalable meaning that it can still be easy to manage even if it spans many independent administrative organizations. Unfortunately, a system that is scalable in one or more of these dimensions often exhibits some loss of performance as the system scales up. Examples of scalability limitations of a distributed system includes –Centralized services (A single server for all users), Centralized data (A single online telephone book), Centralized algorithms (Doing Routing based on Complete information).

A good distributed system can be designed by keeping the following properties in mind which include: reliability, security, heterogeneity, and topology of the network; latency and bandwidth; transport costs; and finally administrative domains.

Types of Distributed Systems: They are divided into distributed computing systems, distributed information systems, and distributed embedded systems. Distributed computing systems are further divided into Cluster computing systems and Grid computing systems.

The Cluster computing systems are used for parallel programming in which a single program is run in parallel on multiple machines. Each cluster consists of a collection of compute nodes that are controlled and accessed by means of a single master node. The master typically handles the allocation of nodes to a particular parallel program, maintains a batch queue of submitted jobs, and provides an interface for the users of the system. The master actually runs the middleware needed for the execution of programs and management of the cluster through remote access, while the compute nodes all have the same operating system, and are all connected through the same standard high speed network. The middleware is a collection of programs called libraries which are meant for executing parallel programs. Many of these libraries effectively provide only the advanced message-based communication facilities, but are not capable of handling faulty processes, security, etc.

The Grid computing systems have a high degree of heterogeneity: no assumptions are made to which their hardware, operating systems, networks, administrative domains, and security policies are concerned. Resources from different organizations are brought together to allow the collaboration of a group of people or institutions. Such collaboration is realized in the form of a virtual organization. The people belonging to the same virtual organization have access rights to the resources that are provided to that organization. Typically, resources consist of compute-servers, storage facilities, databases and special networked devices such as telescopes, sensors etc. The grid computing systems concentrates on providing access of resources from different

administrative domains to only those users and applications that belong to a specific virtual organization.

In Distributed Information Systems, a networked application simply contains a server running an application (often including a database) and making it available to remote programs, called clients. Such clients could send a request to the server for executing a specific operation, after which a response would be sent back. Integration at the lowest level would allow clients to wrap a number of requests, possibly for different servers, into a single larger request and make it executed as a distributed transaction. The key idea was that all or none of the requests would be executed within a single request consisting of various sub-queries. As applications gradually separated into independent components (notably distinguishing database components from processing components), it became clear that integration should also take place by letting applications communicate directly with each other. This leads to a huge industry called Enterprise Application Integration (EAI). The Distributed Information Systems are divide into Transaction Processing Systems and Distributed Pervasive Systems.

In Transaction Processing Systems, a transaction may contain some primitives which include BEGIN_TRANSACTION, END_TRANSACTIUON, READ, WRITE, ABORT_TRANSACTION, etc. The exact list of primitives depends on what kinds of objects are being used in the transaction. In a mail system, there might be primitives to send, receive, and forward mail. In an accounting system, there might be quite different primitives such as READ and WRITE. Sometimes a transaction may include an RPC (Remote Procedure Call) inside of its ordinary statements. RPCs are procedure calls to remote servers, are often also encapsulated in a transaction, leading to what is known as a transactional RPC. As the popularity of object technology increased, techniques were developed to allow calls to remote objects, leading to what is known as remote method invocations (RMI). An RMI is essentially the same as an RPC, except that it operates on objects instead of applications. RPC and RMI have the disadvantage that the caller and called programs both need to be up and running at the time of communication. In addition, they need to know exactly how to refer to each other. This tight coupling is often experienced as a serious drawback, and has led to what is known as message-oriented middleware (MOM). In this case, applications simply send messages to logical contact points, often described by means of a subject. Likewise, applications can indicate their interest for a specific type of message, after which the communication middleware will take care that those messages are delivered to those applications. These so-called publish/subscribe systems form an important and expanding class of distributed systems.

The distributed systems becomes instable because of the inclusion of the distributed pervasive systems which are often characterized by being small, battery-powered, mobile, and having only a wireless connection, etc. At best, such devices can be configured by their owners (for example, a smart phone owner), but otherwise they need to automatically discover their environment and nestle-in as best as possible. This nestling-in has been made more precise by formulating the three requirements for pervasive applications: *Embrace contextual changes* (A device must be continuously be aware of the fact that its environment may change all the time. One

of the simplest changes is discovering that a network is no longer available, for example, because a user is moving between base stations. In such a case, the application should react, possibly by automatically connecting to another network, or taking other appropriate actions.), *Encouraging ad hoc composition* (Many devices in pervasive systems will be used in very different ways by different users. As a result, it should be easy to configure the suite of applications running on a device, either by the user or through automated but controlled interposition.), *Recognize sharing as the default* (One very important aspect of pervasive systems is that devices generally join the system in order to access information, meaning that the pervasive system should be in a position to easily read, store, manage, and share information.).

In the presence of mobility, devices should support easy and application-dependent adaptation to their local environment. They should be able to efficiently discover services and react accordingly. It should be clear from these requirements that distribution transparency is not really in place in pervasive systems. Examples of pervasive systems include Home Systems, Electronic Health Care systems, and Sensor Networks.

Home Systems generally consist of one or more personal computers with wireless integration of consumer electronics such as TVs, audio and video equipment, gaming devices, smart phones, PDAs, etc into a single system. In addition, it is to be expected that all kinds of devices such as kitchen appliances, surveillance cameras, clocks, controllers for lighting, and so on, will all be hooked up into a single distributed system. An important fact here is that such a system should be completely self-configuring and self-managing. A more attention is required in the areas -how software and firmware in devices can be easily updated without manual intervention, or when updates do take place only if the compatibility with other devices is not violated. Another issue to be focused is managing what is known as a "personal space." Recognizing that a home system consists of many shared as well as personal devices, and that the data in a home system is also subject to sharing restrictions, much attention is paid to realizing such personal spaces. For example, part of Sita's personal space may consist of her agenda, family photos, a diary, music and videos that she bought, etc. These personal assets should be stored in such a way that Sita has access to them whenever appropriate. Moreover, parts of this personal space should be (temporarily) accessible to others, for example- When she needs to make a business appointment. Pervasive home systems adopt an architecture in which a single machine acts as a master and is hidden away somewhere in the basement by configuring with a required hard disk capacity and all other fixed devices simply provide a convenient interface for humans. Personal devices will then be packed full with daily needed information, but will never run out of storage. However, having enough storage does not solve the problem of managing personal spaces. Being able to store huge amounts of data shifts the problem to storing relevant data and being able to find it later. Now days, pervasive systems, equipped with what are called recommender-programs that consult what other users have stored in order to identify similar taste, and from that subsequently derive which content to place in one's personal space.

Electronic Health Care Systems- With the increasing cost of medical treatment, new devices are being developed to monitor the well-being of individuals and to

automatically contact physicians when needed. In many of these systems, a major goal is to prevent people from being hospitalized. Personal health care systems are often equipped with various wireless sensors organized in a body-area network (BAN). The network should be able to operate while a person is moving, with no wires attached. There are 2 scenarios made here. First scenario - a central hub is part of the BAN and collects data as needed. From time to time, this data is then offloaded to a larger storage device. The advantage of this scheme is that the hub can also manage the BAN. In the second scenario, the BAN is continuously hooked up to an external network, again through a wireless connection, to which it sends monitored data and further connections to a physician or other people may exist as well.

A lot of work needs to be done in this area by looking into some important perspectives like - Where and how should monitored data be stored?, How to prevent loss of crucial data?, What infrastructure is needed to generate and propagate alerts?, How can physicians provide online feedback?, How can extreme robustness of the monitoring system be realized?, What are the security issues and how can the proper policies be enforced? For reasons of efficiency, devices and body-area networks will be required to support in-network data processing, meaning that monitoring data will, have to be aggregated before permanently storing it or sending it to a physician. Unlike the case for distributed information systems, there is yet no clear answer to these questions.

Sensor Networks- Many solutions of sensor networks return in pervasive applications. They are used for processing information. In this sense, they do more than just provide communication services. A sensor network typically consists of tens to hundreds or thousands of relatively small nodes, each equipped with a sensing device. Most sensor networks use wireless communication, and the nodes are often battery powered. The relation with distributed systems can be made clear by considering sensor networks as distributed databases. To organize a sensor network as a distributed database, there are essentially two extremes to be considered. First, sensors do not cooperate but simply send their data to a centralized database located at the operator's site. The other extreme is to forward queries to relevant sensors and to let each compute an answer, requiring the operator to sensibly aggregate the returned answers. Neither of these solutions is very attractive. What is needed are facilities for in-network data processing. This can be done in numerous ways. One obvious way is to forward a query to all sensor nodes along a tree encompassing all nodes and to subsequently aggregate the results as they are propagated back to the root, where the initiator is located. Aggregation will take place where two or more branches of the tree come together. As simple as this scheme may sound, the following queries need to be answered properly in the implementation part. They are: How to dynamically set up an efficient tree in a sensor network? How does aggregation of results take place? Can it be controlled? What happens when network links fail?

Threads: To execute a program, an operating system creates a number of virtual processors, each one for running a different program. To keep track of these virtual processors, the operating system has a process table, containing entries to store CPU register values, memory maps, open files, accounting information, privileges, etc. A process is often defined as a program in execution, that is, a program that is currently being executed on one of the operating system's virtual processors. Multiple

processes may be concurrently sharing the same CPU and other hardware resources are made transparent by the operating system. Like a process, a thread executes its own piece of code, independently from other threads. A thread context often consists of nothing more than the CPU context, along with some other information for thread management. For example, a thread system may keep track of the fact that a thread is currently blocked on a mute variable, so as not to select it for execution.

Replication and Consistency: An important issue in distributed systems is the replication of data. Data are generally replicated to enhance reliability or improve performance. One of the major problems is keeping replicas consistent. Informally, this means that when one copy is updated we need to ensure that the other copies are updated as well; otherwise the replicas will no longer be the same.

Virtualization: Virtualization can help the legacy interfaces by porting them to the new platforms and thus immediately opening up the latter for large classes of existing programs. Using Virtualization, the diversity of platforms and machines can be reduced by essentially letting each application run on its own virtual machine, possibly including the related libraries and operating system, which, in turn, run on a common platform.

Code Migration: Code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another, simplifying the design of a distributed system. Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so. That reason has always been performance. The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines. Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well. A special care must be taken while performing code migration in heterogeneous systems.

Synchronization: It is important that multiple processes do not simultaneously access a shared resource, such as printer, but instead cooperate in granting each other temporary exclusive access. Another example is that multiple processes may sometimes need to agree on the ordering of events, such as whether message ml from process P was sent before or after message m2 from process Q.

C**onclusion:**

In this paper, all the objectives of distributed systems are clearly mentioned and the areas where proper insight is required for research and implementation are discussed. The technical terms are clearly explained for better understanding of this subject. The basic understanding about distributed systems is given in simple terms which would be really useful to future researchers of this area to proceed with further.

References:

[1] Distributed Sytems- Principles and Paradigms by Tanenbaum, Van Steen

[2] Distributed Systems –An Algorithmic Approach by Sukumar Ghosh

[3] Distributed Sytems -Concepts and Design by Coulouris, Dollimore, Kindberg, Blair

[4] Operating System Concepts – Silberschartz, Galvin, Gagne