Distributed Operating System

RuchiYadav&Pramod Kumar
Dept. of Information & technology, Dronacharya College of Engineering
Farruhknagar, Gurgaon, India
Email:ruchiyadav477@gmail.com

Abstract

A distributed operating system is a control program running on a set of computers that are interconnected by a network. This control unifies the different computers into a single integrated compute and storage resources. Depending on the facilities it provides, a distributed operating system is classified as general purpose, real time, or embedded. The need for distributed operating systems stems from rapid changes in the hardware environment in many organizations. As the price of CPU chips continues to fall rapidly, it will soon be economically feasible to build computer systems containing a large number of processors. In this paper a logical model of a distributed operating system has been presented. This model of a distributed operating system contains a set of processes managing resources, connections between these processes, and mappings of events controlling this distributed operating system into processes managing resources. This article presents a paradigm for structuring distributed operating systems, the potential and implications this paradigm has for users and research directions for future.

1. Introduction

In the last few decades, we've seen computers move from large, monolithic machines that allowed a single user complete access to the entire machine to large machines that allowed multiple users to have access to the machine simultaneously. Then, a decade or so ago, the next step was taken; the machines became smaller and again returned to single user computers, this time being called 'personal computers'. The final step, so far, is to tie these personal computers to a central resource system for shared disks, printers, CPU cycles, and so on; distributed computing.

Since the late 1970's distributed computing, and more specifically distributed operating systems research, has yielded an impressive amount of excellent work, moving to the forefront of computer science research areas in the university environment. The cost of CPU chips is expected to continue declining during the coming decade, leading to systems containing a large number of processors. Connecting all these processors using standard technology (e.g., a LAN) is easy. The hard part is designing and implementing software to manage and use all of this computing power in a convenient way. In this paper we describe a distributed operating system.

2. Two Paradigm

Operating system structures for a distributed environment follow one of two paradigms: message based or object based. Message based operating systems place a message-passing kernel on each node and use explicit messages to support interprocesses communication. The kernel supports local communication and remote communication, which is sometimes implemented through a separate network-manager process. In a traditional system such as UNIX, access to system services via procedure call, whereas in a message based operating system, the requests are via message passing. Message based operating systems are attractive for structuring operating systems because the

*

policy, which is encoded in the server processes, is separate from the mechanism implemented in the kernel.

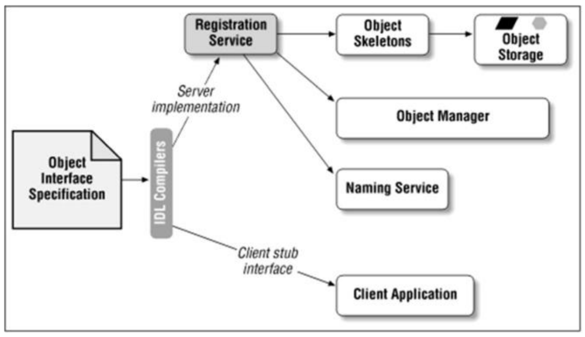
2.1. Message Passing

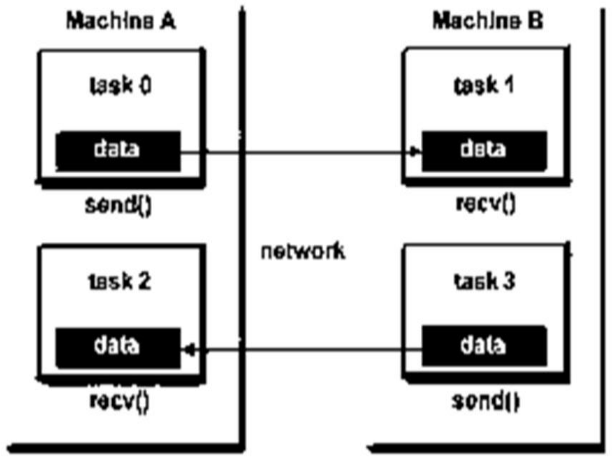
The model that is favored by researchers in this area is the client-server model, in which a client process wanting some service (e.g., reading some data from a file) sends a message to the server and then waits for a reply message. The system just provides two primitives: SEND and RECEIVE. The SEND primitive specifies the destination and provides a buffer; the RECEIVE primitive tells from whom a message is desired (including "anyone") and provides a buffer where the incoming message is to be stored. No initial setup is required, and no connection is established, hence no teardown is required.

Two of the fundamental decisions that must be made are unreliable vs. reliable and no blocking vs. blocking primitives. At one extreme, SEND can put a message out onto the network and wish it good luck. No guarantee of delivery is provided, and no retransmissions are attempted by the system. At the other extreme, the SEND can handle lost messages, retransmissions, and acknowledgements internally, so that when SEND terminates, the program is sure that the message has been received and acknowledged.

The other choice is between non-blocking and blocking primitives. With non-blocking primitives, SEND returns control to the user program as soon as the message has been queued for subsequent transmission (or a copy made). If no copy is made, any changes the program makes to the data before or (heaven forbid) while it is being sent, are made at the program's peril. When the message has been transmitted (or copied to a safe place for subsequent transmission), the program is interrupted to inform it that the buffer may be reused. The corresponding RECEIVE primitive signals a willingness to receive a message, and provides a buffer for it to be put into. When a message has arrived, the program is informed by interrupt. The advantage of these non-blocking primitives is that they provide the maximum flexibility: programs can compute and perform message I/O in parallel any way they want to.

Another design decision that is closely related to the ones above is whether or not to buffer messages. The simplest strategy is not to buffer. When a sender has a message for a receiver that has not (yet) executed a RECEIVE primitive, the sender is blocked until a RECEIVE has been done, at which time the message is copied from sender to receiver. This strategy is sometimes referred to as a rendezvous, and it provides for simple flow control.





*

2.2. Object Based

Object based distributed operating systems encapsulate services and resources into entities called objects. Objects are similar to instances of abstract data types. They are written as individual modules composed of the specific operations that define the module interfaces. Clients request access to system services by invoking the appropriate system object. The invocation mechanism is similar to protected procedure call. Objects encapsulate functionality much as server processes do in message based systems. Most object based systems are built on top of an existing operating system, typically UNIX. Examples of such systems include Argus, Cronus and Eden.

These systems support objects that respond to invocations sent via the message passing mechanisms of UNIX. Mach is an operating system with distinctive characteristics. A UNIX compatible system built to be machine independent; it runs on a large variety of uniprocessors and multiprocessors. It has a small kernel that handles the virtual memory and process scheduling, and it builds other services on top of the kernel. Mach implements mechanisms that provide distribution, especially through a facility called memory objects, for sharing memory between separate tasks executing on possibly different machines.

3. Remote Procedure Call

RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

RPC makes the client/server model of computing more powerful and easier to program. When combined with the ONC RPCGEN protocol compiler clients transparently make remote calls through a local procedure interface.

3.1. How RPC Works

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. **The** low of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

A remote procedure is uniquely identified by the triple: (program number, version number, procedure number) the program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available

*

simultaneously. Each version contains a number of procedures that can be called remotely. Each procedure has a procedure number.

4. Features of Distributed Operating System

The main features of a distributed system include:

• **Functional Separation** Based on the functionality/services provided, capability and purpose of each entity in the system.

• • **Inherent distribution** Entities such as information, people, and systems are inherently distributed. For example, different information is created and maintained by different people. This information could be generated, stored, analyzed and used by different systems or applications which may or may not be aware of the existence of the other entities in the system.

• **Reliability** Long term data preservation and backup (replication) at different locations.

• **Scalability** Addition of more resources to increase performance or availability.

• **Economy** Sharing of resources by many entities to help reduce the cost of ownership.

As a consequence of these features, the various entities in a distributed system can operate concurrently and possibly autonomously. Tasks are carried out independently and actions are co-ordinate at well-defined stages by exchanging messages. Also, entities are heterogeneous, and failures are independent. Generally, there is no single process, or entity, that has the knowledge of the entire state of the system.

Various kinds of distributed systems operate today, each aimed at solving different kinds of problems. The challenges faced in building a distributed system vary depending on the requirements of the system. In general, however, most systems will need to handle the following issues:

• **Heterogeneity –** Various entities in the system must be able to interoperate with one another, despite differences in hardware architectures, operating systems, communication protocols, programming languages, software interfaces, security models, and data formats.

• **Transparency-** The entire system should appear as a single unit and the complexity and interactions between the components should be typically hidden from the end user.

• **Fault tolerance and failure management** Failure of one or more components should not bring down the entire system, and should be isolated.

• **Scalability**

The system should work efficiently with increasing number of users and addition of a resource should enhance the performance of the system.

• **Concurrency** Shared access to resources should be made possible.

• **Openness and Extensibility**

Interfaces should be cleanly separated and publicly available to enable easy extensions to existing components and add new components.

• **Migration and load balancing –** Allow the movement of tasks within a system without affecting the operation of users or applications, and distribute load among available resources for improving performance.

• **Security** Access to resources should be secured to ensure only known users are able to perform allowed operations.

Several software companies and research institutions have developed distributed

*

computing technologies that support some or all of the features described above.

5. Cloud Approach

A Cloud is a distributed operating system that integrates a set of nodes into a conceptually centralized system. The system is composed of computer servers, data servers, and user workstation. A compute server is a machine that is available for use as a computational engine. A data server is a machine that functions as a repository for long-lived (that is persistent) data. A user work station is a machine that provides a programming environment for developing applications and an interface with the computer and data servers for executing those applications on the servers.
Note that when a
disk is advocated with a computer server. It can also set as a data server for other compare servers. A cloud is a native kernel called Ra (after the Egyptian sun god). It currently runs on sun-3/50 and sun-3/60 computers an cooperators with sun spare stations (running units) that provide user interfaces.

6. Conclusions

The paper presents a comprehensive summary of the ideals of a distributed operating system. By having a microkernel, most of the key features are implemented as user processes, which mean that the system can evolve gradually as change and we needs learn more about distributed computing. The object-based nature of the system and the use of capabilities provide a unifying theme that holds the various pieces together. We believe the parts of our RPC package here discussed are of general interest in several ways. They represent a particular point in the design spectrum of RPC. We believe that we have achieved very good performance without adopting extreme measures, and without sacrificing useful call and parameter semantics.

Cloud computing is a major development in information technology, comparable in importance with the mainframe, the minicomputer, the microprocessor, and the Internet. It has the potential to make an increasingly significant contribution to economic activity throughout the world. This potential will only be realized if cloud computing products and services are portable and interoperable.

7. References

[1] G. Couloirs, J. Dollimore, and

T. Kinberg, Distributed Systems - Concepts and Design, 4th Edition, Addison-Wesley, Pearson Education, UK, 2001.

[2] R. Buyya (editor), High Performance Cluster Computing, Prentice Hall, USA, 1999.

[3] Tanenbaum, A.S., van Staveren, H., Keizer, E.G., and Stevenson, J.W.: "A Practical Toolkit for Making Portable Compilers," Commun. ACM, vol. 26, pp. 654-660, Sept. 1983.

[4] R. Subramanian and B. Goodman (editors), Peer-to-Peer Computing: Evolution of a Disruptive Technology, Idea Group Inc., Hershey, PA, USA, 2005.

*