A Comparison of Distributed Systems: ChorusOS and Amoeba

Angelo Bertolli
Prepared for MSIT 610 on October 27, 2004
University of Maryland University College
Adelphi, Maryland
United States of America

Abstract. Distributed systems provide a framework to take advantage of the under used computing power of idle machines, and to attempt to solve problems too difficult for ordinary machines. This paper will attempt to define distributed computing and demonstrate the strengths, weaknesses, techniques, and theories behind distributed systems. The scope will include an introduction and discussion of distributed systems and a look at two specific systems: ChorusOS and Amoeba.

Section 1: General Overview of Distributed Systems

Introduction

Flynn's taxonomy attempts to categorize computer systems in terms of how they use data, and how that data is controlled or manipulated. It defines two control streams: SI (single instruction) and MI (multiple instruction), and two data streams: SD (single data) and MD (multiple data). Any computer system will have one type of control stream, and one type of data stream:

		Control	
		Single	**Multiple**
Data	**Single**	SISD A sequential computer	MISD *Unusual**
	Multiple	SIMD Vector Processors Massively Parallel Processors	MIMD Distributed Systems

* "There are few machines in this category, none that have been commercially successful or had any impact on computational science." [3]

The focus of this paper will be on distributed systems: those computer systems which use multiple control streams upon multiple data streams (MIMD). [10] "The category of MIMD machines is the most diverse of the four classifications in Flynn's taxonomy." [3]

Distributed systems provide a large amount of computing power by distributing the work load across smaller, less-capable machines. This means that it is less expensive to build a distributed system than to create a new autonomous system which can do the same job. This is typically done with a model called *Message Passing* in which a program is split into smaller pieces which can pass information to each other.

Applications of distributed systems include particle physics, bioinformatics, office resource sharing, visualization & graphics rendering, enterprise management, and embedded systems (like your cell phone). Some issues concerning the design of distributed systems include fault tolerance, process scheduling, and security.

Categorization

Distributed systems can be divided into two main categories: homogeneous systems and heterogeneous systems. These categorizations are based on function within the system, and sometimes on the types of hardware that make up the system.

	Examples
Homogeneous	■ Processor Pool: connected workstations.
	■ Cluster: hardware facility for fault tolerance.
Heterogeneous	■ Client/Server: one server machine (bigger, with shared resources), and multiple client machines (smaller).
	■ Network of Workstations (NOWs): similar to client/server, but with more equal computing power—can be over either a LAN or WAN.

Distributed systems can also be categorized according to their software environments (in order of decreasing abstraction from the user):

(1) **Distributed Operating System:** completely manages the hardware of multiple individual nodes, as to appear as one meta-node.

(2) **Distributed Shared Memory Systems:** manages the memory and processor usage of the different nodes.

(3) **Network Operating System:** individual systems with network capabilities, linked to effectively provide a distributed system.

Speed-up and Efficiency

The critical factor in determining the usefulness of a distributed system is its speed-up or efficiency: how much better multiple processors can solve a problem than one processor. Even without overhead due to communication, no distributed system can run at 100% efficiency (the speed-up is equal to the number of processors). This is because there is always a percentage of sequential code which cannot be done parallel.

Let R represent the total run time of the program.
Let R_S represent the run time of the sequential code. ($R_S > 0$)
Let R_P represent the run time of the code done in parallel (on one processor).

Therefore, our total run time will be the sum of the sequential part of the problem, and the parallel part of the problem:

$$R = R_S + R_P$$

If we consider n to represent the number of processors, each processor will share the part done in parallel. We have:

$$R = R_S + (R_P / n)$$

(Again, this is considering there is no loss due to overhead for communication.)

The speed-up of a system is defined as the ratio of the system with a single processor to the same system with n processors:

$$S = (R_S + R_P) / (R_S + (R_P / n))$$

And since R_S can never be zero, you cannot achieve a speed-up equal to the number of processors. A speed-up equal to the number of processors means an efficiency of 100%. Efficiency is defined as the speed-up divided by the number of processors:

$$E = S / n$$

Example

With an algorithm that is 20% sequential, 80% parallel, and requires 100 seconds to run on a single processor (R), we can determine the speed-up and efficiency of using 4 processors (R_4) (assuming no overhead):

$R = 100$ (seconds) $= R_S + R_P$
$R_S = 20$
$R_P = 80$
$R_4 = R_S + (R_P / n) = 20 + (80 / 4) = 40$
$S = R / R_4 = 100 / 40 = 2.5$
$E = S / n = 2.5 / 4 = 0.625$

In this case, our speed-up of four processors is 2.5 times more than of one processor, or an efficiency of 62.5%

Software for Distributed Programming

There are a number of software packages which have been created to make writing programs for distribution over many machines easier. These packages have some way of transporting data or services among nodes in the system. Three popular packages include PVM, RPC, and CORBA.

PVM (Parallel Virtual Machine) – This software package was designed to provide a programming environment that emulates a traditional parallel system.

RPC (Remote Procedure Call) – This software package provides procedures or functions which can be accessed remotely. When the procedure call is made by the first node, data is sent to the second node offering the function, the function is executed, and the results are returned to the first node.

CORBA (Common Object Request Broker Architecture) – This software package is similar to RPC in architecture, but instead of providing functions, CORBA provides C++ objects to other nodes. Then the first node requests the use of an object offered by the second node, the second node creates the instance of the object locally. All commands issued to the object by the first node occur on the second node.

Correct Distributed Systems

It is useful to describe a system with more detail, beyond the generic term of just "distributing the work." Correct distributed systems (called truly distributed) are defined to have features which not only provide enhanced security and fault tolerance, but a desired level of abstraction to the user. They have four main characteristics:

(1) That the relevant information is going to be distributed among the nodes.

(2) That each process in the distributed system can make decisions given information which is local to that node.

(3) There is no common point of failure. For example, in a homogeneous system, any 1/3rd of all nodes should be able to go down without there being a noticeable difference except for performance loss.

(4) A truly distributed system cannot rely upon a global clock to function.

One alternative to a global clock is to use the Lamport Clock Algorithm for a logical clock. A logical clock defines the operation "happens before", e.g. "a happens before b" A node using the Lamport Clock Algorithm can keep track of time by attaching a send time to each message, and increment its local clock according to the messages it received.

Section 2: Descriptions of Specific Systems

ChorusOS

Chorus is an operating system for embedded systems. An embedded system is a specialized system that is part of a larger system or machine. Chorus is a real-time OS (RTOS) which means that it responds immediately to input and can be used in real-time applications to solve real-time problems.

At its heart is the Chorus nucleus (microkernel). A microkernel is an OS with only the essential services such as interprocess communication, short-term scheduling, and memory management. Chorus implements these simple tools and makes them distributed on the lowest level.

Chorus is designed for reliability, availability, and scalability for critical telecommunications applications.

Background

Chorus started in 1980 as a research project at the French research institute INRIA. [7] It started with version 0 which implemented structured processes called "actors." In 1982, version 1 continued with multi-processor research and included structured messages and support for fault tolerance. Version 2 came out in 1984 and was source compatible with UNIX, meaning that UNIX programs could be recompiled and run on Chorus. In 1987, with version 3, Chorus went commercial, becoming Chorus Systems Inc. Furthermore, version 3 implemented RPC as the communication model.

Chorus Cool ORB 3.1 became available in 1996. This ORB is designed for distributed, real-time application development, and is targeted at telecommunication companies. It allows companies to create applications that run on different platforms simultaneously over a network. Chorus Cool ORB also complies with CORBA 2.0 object specifications.

In 1998 Sun Microsystems acquired Chorus, and since then they have released Chorus 4.0.1

Chorus Microkernel

The Chorus microkernel (which they like to call the "nucleus") has 4 major parts. The first part, the supervisor, handles hardware interrupts, traps, and exceptions. [7] This portion must be rewritten when it is ported to a new system. The second part, the real-time executive, manages the processes, threads, and scheduling, along with the synchronization between threads. The virtual memory manager takes care of the low-level part of the paging system. The inter-process communication manager handles the UI's (unique identifiers), ports, and sending messages in a transparent way.

The Chorus microkernel has low-level transparent connectivity services that let you distribute functions across multiple PCs. When one fails, you can automatically off-load to another. The size of the Chorus microkernel is about 50 to 60 Kb.

Processes

There are 3 kinds of processes which interact with each other and the nucleus to get jobs done. Starting with the lowest level, the processes are kernel, system, and user. Kernel processes reside in kernel space along with the nucleus. System and user processes reside in user space.

- Kernel processes are trusted and privileged, able to make demands on the nucleus and implement their own hardware functions. Kernel processes complete the necessary requirements for a basic operating system, and provide things such as file managers, stream and sock managers, device drivers, etc. However, these processes are only loaded as needed and can be removed or added during system execution. The relationship between kernel processes and the nucleus provide a way to extend and configure the functionality of the OS.

- System processes are trusted, able to send requests to kernel processes and to the nucleus itself. A subsystem consists of a collection of system processes that work together, eg. the UNIX subsystem known as MiX allows binary compatibility with UNIX. [7]
- User processes are neither trusted or privileged. They may only ask something of the nucleus or kernel through a subsystem. No direct calls are allowed.
- Real-time processes can run as system processes to reduce overhead.

Amoeba

The Amoeba operating system is designed to run on multiple machines across a network which appears to each user as a single shared machine. [9] It is a heterogeneous distributed operating system designed for general-purpose computing. "It is designed to take a collection of machines and make them act together as a single integrated system." [8] Details about where processes are run or files are being stored are hidden to the user. Amoeba can be used both as a distributed system for multiple users with multiple tasks, or a parallel system for a single difficult job.

Background

There was an explosion in personal computing in the 1980's when computers became affordable enough for each user to have a workstation. Around this time, a group at the *Vrije Universiteit* (VU) in Amsterdam started working on the problem of using multiple individual user systems in distributed and parallel ways.

System Design

The design goals for the Amoeba operating system included distribution, parallelism, transparency, and performance:

Distribution – Amoeba is a heterogeneous system which can be built on a LAN.

Parallelism – Amoeba can use multiple processors for a single job to decrease the total run time for the job.

Transparency – Amoeba hides details about how the resources of the various machines are handled from the user.

Performance – Amoeba uses an optimized communication system to reduce the overhead associated with distributed and parallel computing.

Amoeba also uses a microkernel architecture in which a copy runs on each node in the system. This kernel supports only the basics needed locally for that system such as minimal process and communication primitives, memory management, and raw device I/O. Server processes which run on top of the microkernel are usually in user space.

The main communication mechanism for Amoeba is *remote procedure call* (RPC). This is also hidden from the user in the *Amoeba Interface Language* (AIL) which provides functions and procedures that handle the details of communication.

Filesystem

Amoeba has dedicated file servers called *Bullet servers.* These servers store only the file data and not the names or hierarchy of the filesystem. "When a user program needs a file, it will request that the Bullet server send it the entire file in a single RPC." [8] This means that a file server must have at least 16 MB of RAM, and the amount of physical memory limits the maximum file size. Directories and file names are handled by separate directory servers.

TCP/IP

Amoeba doesn't use TCP/IP for it's internal communication. However, an Amoeba system can contain a special server to provide TCP/IP communication to other systems.

"Amoeba is available for free to universities and other educational institutions and for special commercial prices and conditions to corporate, government, and other users." [8]

Section 3: Comparisons and Conclusions

Amoeba was designed to be used on multiple CPU's connected by a LAN, whereas Chorus was designed to provide distributed services and architecture for embedded systems. Chorus started out far from UNIX, but has increasingly become closer and closer to UNIX. While Amoeba is designed to abstract or hide the multiple machines from users, Chorus was designed with the idea that users would not only be logging into a particular machine but would be doing most of their work on that machine.

Item	**Amoeba**	**Chorus**
Designed for:	Distributed system	1 CPU, multiprocessor
Automatic load balancing?	Yes	No
Threads managed by:	Kernel	Kernel
Transparent heterogeneity?	Yes	No
Multiprocessor support?	Minimal	Moderate
UNIX emulation	Source	Binary
UNIX compatibility	POSIX (partial)	System V
Automatic file replication?	Yes	No

This is a partial chart from [8] p517: **Fig. 9-28.** A comparison of Amoeba, Mach, and Chorus

References

1. Abrossimov, Armand, Boule, Gien, Guillemont, Herrman, Kaiser, Langlois, Leonard, and Neuhauser. *Overview of the Chorus Distributed Operating System.* http://www.cs.berkeley.edu/~gribble/osprelims/summaries/Chorus.html.
2. Borzo, Jeannette. "Chorus System ships real-time application development ORB for telcos." *InfoWorld.* August 26, 1996: p52
3. Computational Science Education Project. *Computer Architecture.* Copyright © 1991, 1992, 1993, 1994, 1995. http://csep1.phy.ornl.gov/ca/ca.html: Section 3.1
4. Foley, Mary Jo. "SCO, Chorus prep microkernel Unix for real-time apps." *PC Week.* December 6, 1993: p65
5. Podesta, Karl. *Distributed Systems (A very basic introduction of real world examples).* http://www.computing.dcu.ie/~kpodesta/distributed/.
6. Pountain, Dick. "The Chorus Microkernel." *Byte.* January 1994: p131-132
7. Tanenbaum, Andrew S. *Distributed Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 1995.
8. Tenenbaum, Andrew S., and Sharp, Gregory J. *The Amoeba Distributed Operating System.* Vrije Universiteit, De Boeleaan 1081a, Amsterdam, The Netherlands.
9. Wikipedia, the free encyclopedia. *Amoeba distributed operating system.* http://en.wikipedia.org/wiki/Amoeba_distributed_operating_system
10. Wikipedia, the free encyclopedia. *Flynn's taxonomy.* http://en.wikipedia.org/wiki/Flynn's_taxonomy
11. Williams, Tom. "Sun Microsystems maps its embedded systems strategy with Java." *Electronic Design.* January 26, 1998: p90
12.