Distributed Operating Systems

Sebastian Kropp

31 August 2004
Monash University
Faculty of Information Technology
s.kropp@microlution.de
Student ID: 19363915
Caulfield, VIC

Abstract

The intention of this writing is to introduce the reader to problems, requirements and design issues of distributed operating systems. After introducing the main motivations, which lead to distributed operating systems and defining what is meant by a distributed operating systems, this paper takes a deeper look into what transparency, reliability, flexibility and performance of those systems imply. The last sections are covering process allocation and real-time constraints. This work is mainly based on the book "Distributed Operating Systems" [1] by A. Tannenbaum in 1995. Since then the environment for distributed systems has changed and new approaches are reviewed.

Contents

1 Introduction

The inherent evolution of more powerful processors does not to satisfy the demand for computation. Most of those computations are highly dependant on time. We do not want to wait for an email to arrive at the same time as normal mail would or a certain simulation to finish till a competitor has already done it and started to ship out the resulting product. As we can see from those examples, the computational power needs to meet commercial and time constraints. Not the overall average in computational power is important, but how deadlines require peak times and how the system scales to such events. Although there are examples like the SETI program where response time does not play the crucial role, those applications are seldom. In applications where a missing of a deadline directly or indirectly leads to a human catastrophe, time is a hard constraint. Those hard real-time constraints are getting more important as computer assist more and more our daily life. Two approaches are taken to meet the demand for computations. One is the steady improvement of hardware. CPU improve in an incredible speed, but to do so is quite costly and meets physical barriers. The second one tries to relax this situation by working with more than one CPU on a problem in parallel. It is useful to distinguish two kinds of CPU interconnections in which the response time and data transfer rate are the differing factors. Loosely coupled CPUs are those, which work together and possible communication is slow, distant and more unreliable. This setup called multicomputer and make up distributed systems. An example would be computers connected via modem or LAN. Multiprocessors on the other hand are closely coupled CPUs with a high bandwidth and response time often working on a shared memory. The communication is based on a multi processor board. These systems are not considered to be distributed systems, although it is not uncommon that they are used in a distributed environment. The price/service ratio between mainframes and PC workstations has changed. Having 10 machines with 100 MIPS than one with 1000 MIPS has become more economical. But combining these 10 computers puts a burden on the system design. Certain parallel algorithms and computational problems require high constant interaction of CPUs and other do not.

Some systems are inherently distributed. An example is the inventory and store management of a company. Stores are located all over the country and loosely interconnected with each other to be able to perform global optimization. It is hardly feasible to centralize this structure.

Another motivation for distributed systems is reliability. The probability of the system to fail is decreased by involving more than one computer. This redundancy requires the transparency that the different nodes in a cluster are able to take over the work of a failing machine and only jeopardize the quality of service. Developing software is the most complex issue in distributed systems. The correctness of concurrent working processes, the detection of faulty nodes and timing constraints in the network impose serious problems for the system designer. Distributed operating systems should be able to assist the design process by keeping complex tasks of synchronization, optimal process allocation and redundancy away from the programmers. So transparency to the programmer is another requirement for feasible distributed systems. Hiding complexity by transparency becomes necessary in such complex systems.

Although these issues are known for a long time, modern operating systems have huge problems accommodating changes towards a distributed environment. Traditionally single CPU computers did not require complex synchronization and changing these systems to cope with real parallelism has shown to be greatly complicated since starting the design of new operating systems from scratch is hardly feasible because legacy software is still economical to use. This is another reason for transparency. New designed operating systems tend to tackle the problems with small interconnected modules. These are micro-kernel systems in which each module tries to solve a small part of the big problems. Since development of distributed systems is still an unknown issue, micro-kernels are way more flexible to accommodate changes in the design. This paradigm of modules is expected to creep into monolithic kernels. In the next sections we look at how kernels could manage process allocation and show some problems with that.

2 Different Architectures and Models

Two main different models can be distinguished for sharing CPUs. The workstation and the processor pool model. In the workstation model the cooperating CPUs are spread over the entire network. Diskless workstations can be used to share common data on a file server while processing is done locally. Workstations with disks can be used to increase reliability of persistent data with a transparent distributed file system. This has proven to be hard and no standard distributed file system exists today which solve all problems of such an approach. Consistency in such a configuration is hard to maintain. Some distributed databases exist which handle those problems with transactions. Atomic transactions are also used to synchronize processes.
The multiprocessor model of a processor pool is a centralized approach. X window terminals share a pool of processors at a centralized point. The benefit is scalability. As soon as new computational needs arise, additional processors can easily be plugged into the processor board which is shared evenly among all users. Administration of a consistent environment is easily possible. The main disadvantages of this approach are the introduction of a single point of failure and high cost of mulitprocessors. For those reasons processor pools are not considered to be distributed systems although it might be argued that the X terminals provide computational services to remote locations. Scheduling decisions are far easier in processor pools. The locations of processes in a SMP (Symmetric MultiProcessing) system do not play a crucial role. It is only slightly beneficial to place processes on CPU where they ran on before, to avoid unnecessary cache invalidations on the die. In some multiprocessors, NUMA (non-uniform memory allocation) architecture makes it necessary to place processes on processors close to their memory. But this is far less complex that the considerations in the workstation model where heterogeneous architectures have to work together and communication is likely to brake down.

3 True Distributed Systems

A truly distributed operating system gives the user the illusion to work on a single machine. It is not apparent, that the system the system consists out of several loosely coupled workstations. Current network operating systems do not deliver that. In Windows or Linux the user has to define, on which machines the service should run. These systems are slowly maturing in the SMP mode, where multiproccessors share a single memory space. Often these operating systems still lack efficiency, because global locks where used to make processes run consistently on more than on processor. Since locking the whole kernel memory is not necessary in all cases of synchronization, fine grained locking is just about to get introduced. For multicomputers those systems still lack the single-image illusion. They communicate over standard protocols but a centralized control is remained. For this to disappear, a single distributed system call interface and file system has to emerge. For example all kernels need to cooperate and decide where to run a certain process. Decentralized control poses a huge challenge to the software design and the mathematical model is hardly understood. Two decentralized approaches for process allocation are shown in the section Scheduling Algorithms.
Five aspects of true distributed systems are transparency, flexibility, reliability, performance and scalability:

3.1 Transparency

Achieving true transparency requires fooling both the user and the programmer with a single-image of the system. Location transparency forbids the definition of single machines. Having names specifying a server like /serverhostA/news is not transparent. Migration transparency involves that for example processes are free to move from workstation to workstation with their complete state and no implication for the program. Local resources need to be proxied to the new location in this case. Parallel transparency even requires that the programmer should not be bothered on how to make his algorithms to run in parallel.

Achieving this is the holy grail of distributed systems and no current existing systems even touch this.

3.2 Flexibility

Flexibility is very important for a distributed system since changes in this environment is inherent and not all design decisions yield to a possible system. It should be easy to reverse decisions without losing previous work. Since nobody can argue against flexibility it leaves to define what it means for operating systems. Two concepts exist today. Micro kernel are mainly responsible for sending messages to the user level processes. On the other hand the monolithic kernel tries to solve user program requirements under the security shield of the kernel space. The performance of micro-kernels may be worse since more context switches are required. Also the task of providing security is harder in this system, since the security has to be valid in the user space and can not be hidden in the kernel space. But once security is achieved in micro kernel, it is no problem to extend that to multiple computer. An approach to this solving security issues with capabilities is shown in [2].

3.3 Reliability

The goal of reliability is one of the main reasons for a distributed system. Availability of services is a main aspect. This can be achieved by decoupling dependant services in the design and redundancy. As soon as one machine fails another takes over the task. These services and their usage should be fault-tolerant. The failure of one composed should not bring down the entire program.

3.4 Performance

Transparency and reliability are very expensive in terms of performance. Adding or multiplying integers remotely is never reasonable, since the message overhead eliminates the benefit of doing this remotely. In some cases it is impossible to predict, whether a remote operation is a gain in performance, because the computation may depend on external events.

3.5 Scalability

Considerable amount of design effort has to deal on how to extend a system. Centralized structures like a database which serves all incoming request may soon turn out to be a bottleneck for the whole system. Congestions in network traffic prevent the system to grow further in size. A good example of a scalable system is the DNS. Here database entries and requests to the root servers are highly decentralized in a hierarchical structure. This scalability is paid with the price of very slow changes of entries. It sometimes takes days for changes to propagate, but even Denial of Service attacks have proven to be hard to conduct against the DNS.

4 Scheduling Algorithms

For the consideration of placing the different processes in a truly distributed workstation systems numerous aspects have to be looked at. Attributes of processes like memory needs, processing requirements, floating-point support and communication requirements are among others, crucial for optimal placement on remote machines. Not all of those attributes can be obtained in advance and depend on external events. Hence a completely optimized allocation scheme is not possible and most scheduling algorithms rely on heuristics which try to yield to better overall performance than random allocation. One other problem is that the scheduling algorithms may easily be in P or even NP in complexity which results in a considerable overhead. In some cases it is not even worth it to move the process to a remote location since the involved overhead in doing so might be greater that running the process locally instead. This also can not be calculated prior to the decision. Some simple algorithms are introduced in the following. Most algorithms assume that it is possible to calculate the usage of a CPU. This might not really be

the case in practice though. The kernel needs to disable timer interrupts in some cases, which leads to an improper assessment of time. New processors contain a cycle counter (Time Stamp Counter TSC register for Intel x86), which can be used to tackle that problem.

4.1 Graph Theoretic Deterministic Algorithm

For the scheduling decision a graph is created in which each node refers to a process and the arcs are weighted with the cost of moving that code to another machine. The graph is then broken down into sub graphs which are then placed on the computers of the network. Graph optimization is fairly well established and would lead to an optimal performance. But it can not be used in practice since required information of costs is seldom known in advance.

4.2 Centralized Up-Down

In the up-down algorithm a central table is maintained in which usage other the different workstations are recorded. It assigns the most points to a machine that has the most processes on it. For new processes the workstation with the minimum points gets selected. It is obvious that this approach leads to a suboptimal solution since the number of processes does not determine the load on the processors. The centralized decision also lacks the ability to supply sufficient reliability.

4.3 Market Model for Resource Allocation

This algorithm transfers a market model into the allocation problem. Each process is assigned a certain cost referring to the assumed resource demand of a process. These processes have to bit for allocation on CPUs, while processors determine the cost on demand and available resources. Although this seems to be a good idea several problems have to be dealt with. How are processes earning money, what is about inflation etc. But it has the advantage of being decentralized. Some of those open issues are addressed in the study from A. Messer and T. Wilkinson [3].

4.4 Immune System Response

This is a very promising idea of a population bases meta-heuristic. The immune system is known to be able to adapt very fast to intruding cells and even remember already seen patterns. The intruder is the ready-to-run process in this case. The B-Cells have a remarkable ability to adapt to changes for example in the process patter by brewing solutions in a micro evolutionary process. This exploits both the ability to specialize and generalize at once and can be compared to genetic algorithms and neural networks put together. This idea is very complex and interested readers should refer to [4]. No centralized control is involved, which is another asset and the immune system. It is highly fault-tolerant. These benefits make it interesting for other applications in distributed systems like intrusion detection systems.

There are a lot more algorithms designed to tackle this problem. It should be obvious that none of them will every lead to an optimal allocation in retrospect, but certain algorithms and combinations of them are more suitable for different problems and environments.

5 Safety Critical Systems and Real-time

For safety critical systems and real-time systems other considerations are important. Not the mean average of performance is important but the ability to react to a certain event with an upper bound. Static allocation is suitable for event-triggered systems while dynamic allocation may be more suitable for time-triggered systems. Cooperation and pre-emptive schemes have to be looked at. Combination of those may be beneficial. Usually the problem of redundancy and deterministic reaction time is dealt with additional hardware. One processor might be dedicated to a problem, while the others deal with general

purpose issues. In terms of efficiency this might be suboptimal, but if context switches and interrupts are kept away from the dedicated CPU, it is easy to ensure deterministic behavior.

6 Discussion and conclusions

By now it is clear how nice the benefits of distributed systems are, but it is by far not trivial to design them. Legacy issues and complexity prevent a faster development of required distributed operating systems. Desirable meta-heuristics like Ant Colony Optimization (ACO), Swarms, normal Genetic algorithms and Neural Networks are just being explored and their advantages are not yet clearly understood. Real-time micro kernel systems begin to emerge and reach a reasonable status to provide true transparency to the system designer. Sophisticated micro kernel systems like QNX [5] are commercially available providing the designers with a broad choice of system configuration. Resource allocation is quite remarkable in QNX. The micro kernel modularization offers great flexibility without a significant loss in efficiency and performance. Legacy monolithic kernels are expected to follow this example and distributed systems will result in higher reliability and greater cost efficiency. The micro kernel system designed by A. Tannenbaum called Amoeba [6] at the Vrije University in Amsterdam seems to be not active anymore. As a nice spin-off, the new language called Python [7] has emerged and without it, we probably would not have seen the existence of Google. This shows how much this area is still in a trial and error phase since the early beginnings in 1980ies.
A very active area is the design of development environments. Several Remote Procedure Call systems are available like CORBA, Microsoft .NET and Java RMI. An industry standard for C and C++ for parallel programming is the Parallel Virtual Machine (PVM) [8]. PVM is based on message passing and widely adopted in cluster computing.
It remains to be seen which approaches will remain and how distributed operating systems will look like. But it is commonly agreed upon, that we will see self repairing and balancing operating systems coming alive in the future.

References

[1] A. Tannenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[2] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pages 558–563, Washington, DC, 1986. IEEE Computer Society.

[3] A. Messer and T. Wilkinson. A market model for resource allocation in distributed operating systems, 1995.

[4] Roger L. King, Aric B. Lambert, Samuel H. Russ, and Donna S. Reese. The biological basis of the immune system as a model for intelligent agents. Box 9627, Mississippi State, MS 39762-9627, U.S.A. MSU/NSF Engineering Research Center for Computational Field Simulation.

[5] QNX Software Systems Ltd. Qnx software systems. World Wide Web page [http://www.qnx.com].

[6] Vrije University. Amoeba. World Wide Web page [http://www.cs.vu.nl/pub/amoeba/].

[7] Python Software Foundation Home Page. Python. World Wide Web page [http://www.python.org].

[8] Parallel Virtual Maschine. Pvm. World Wide Web page [http://www.csm.ornl.gov/pvm/pvm_home.html].