DIPC: The Linux Way of Distributed Programming

Kamran Karimi
Department of Computer Engineering
Iran University of Science & Technology
Narmak, Tehran
Iran

E-mail: karimik@sun.iust.ac.ir

Mohsen Sharifi
Software Engineering Laboratory
Department of Computer Engineering
Iran University of Science & Technology
Narmak, Tehran
Iran

E-mail: mshar@vax.ipm.ac.ir

Abstract

Linux is an easily available and powerful operating system, but it is based on a 70s design, making the need for the addition of more modern concepts apparent. This article lists the main characteristics of Distributed Inter-Process Communication (DIPC), a relatively simple system software that provides users of the Linux operating system with both the distributed shared memory and the message passing paradigms of distributed programming. Distributed programs using DIPC can be independent of any physical network characteristics and there is no need for the use of any link libraries or special compilers. DIPC works by making UNIX System V IPC mechanisms (shared memories, messages and semaphores) network transparent, thus integrating neatly with the rest of the system. The underlying network protocol used is TCP/IP. DIPC can work in Wide Area Networks (WANs) and in heterogeneous environments.

Introduction

Before Linux, powerful UNIX operating systems were considered a luxury. Linux made it possible for ordinary people to have access to an affordable and reliable computing platform. The only problem is that Linux was originally based on decades-old designs [7], making it less attractive for more technically minded users. Linux's answer to this problem is in either port and adaptation, or introduction of newer concepts.

Building Multi-Computers [1] and programming them are among the more popular research subjects, and demand for them is rapidly rising. Any solution to distributed programming under Linux should keep up with one of Linux's more important features: availability to ordinary users.

Motivation

Linux already has symmetric multi-processing capabilities. However, it does not provide enough standard facilities for distributed software development. Programmers and users have to resort to different add-on packages and various programming models to write or use distributed software. The mechanisms provided by these packages usually differ greatly from one another, each requiring the users to learn some new material, which won't be of any use to them when migrating to other methods. Many also require detailed involvement of the programmer in the process of transferring data over the network. One example is the PVM (Parallel Virtual Machine) software [8]. The need for a simpler distributed programming model, usable by more programmers, is clearly present.

The DIPC Software

DIPC (Distributed Inter-Process Communications) is a software-only solution for enabling people to build and program multi-computers [1] easily. Each node can be an ordinary personal computer. These nodes should be connected to each other by a TCP/IP [3] network. It does not use network broadcasting, which helps it work in networks with no such capabilities. There is also no assumption of the existence of a synchronized clock. These mean that DIPC could be used in a Wide Area Network (WAN).

Right from the start, it was decided that ease of application programming, and the simplicity of the DIPC itself should be among the most important factors in the system design, even if they meant that there will be some loss in performance. This decision was backed by the observation that computer and telecommunication equipment’s speeds are improving very rapidly, while training and programming times for distributed applications seem not to be following a similar trend.

In DIPC, UNIX System V IPC mechanisms [4], consisting of semaphores, messages and shared memories, are modified to function in a network environment. This means that installing DIPC requires changing and recompiling the kernel. Here the same system calls that are used to provide communication between processes running in the same computer could be used to allow the communication of processes running on different machines. There is no new system call for the application programmers’ use. There is also no library to be linked to the application code, and no need for any modifications in compilers. DIPC could be used with any language that allows access to operating system’s system calls. It is completely *camouflaged* in the kernel.

The above means that DIPC supports both the message passing and the distributed shared memory paradigms of distributed programming, which results in more options for application programmers [5]. Also, allowing the processes to share only selected parts of their address space helps to reduce the problems of false sharing.

It was decided to implement DIPC in the user space as much as possible, with minimal changes to the kernel. This can lead to a cleaner and simpler design, but in a monolithic operating system, such as Linux, has the drawback of requiring frequent copy operations between kernel and user address spaces [2]. As UNIX does not allow user space processes to access and change kernel data structures at will, DIPC has to have two parts: the more important part is a program named *dipcd*, which runs with superuser privileges. dipcd forks several processes to do its work. The other part is inside the kernel, giving dipcd work to do and also letting it to see and manipulate kernel data. The two parts use a private system call to exchange data. This system call should not be used by other processes in the system.

DIPC provides easy data transfer over the network, and assumes that the code to use these data already resides at the suitable places. Then is justifiable when one Considers the fact that in most cases, the programs’ codes change much less frequently than the data they use,

DIPC is only concerned with providing some *mechanisms* for distributed programming. The policies, i.e. how a program is parallelized, or where an application program’s processes should run, are determined by the programmer or the end user.

DIPC Clusters

DIPC enables the creation of *clusters* of PC computers. Computers in the same cluster could work together to solve a problem. DIPC's clusters are logical entities, meaning that they are independent of any physical network characteristics. Computers could be added to or deleted from a cluster without the need to change any of the network parameters. Several clusters may exist in the same physical network, and each computer could belong to at most one of them. Computers on the same cluster may even be connected to each other by a WAN. As far as DIPC is concerned, computers in one cluster never interact with computers in other clusters.

In normal System V IPC, processes specify numerical *keys* to gain access to the same IPC structure [4]. They can then use these structures to communicate with each other. A key normally has a unique meaning only in one computer. DIPC makes the IPC keys globally known. Here, if the application programmer wants it, a key can have the same meaning in more than one machine. Processes on different computers can communicate with each other the same way they did in a single machine.

Information about all the IPC keys in use are kept by one of dipcd's processes called the *referee*. Each cluster has only one referee. In fact, it is having the same referee that places computers in the same cluster. All other processes in the cluster refer to this one to find out if a key is in use. This means that the referee is DIPC's name server. Beside many other duties, the referee also makes sure that only one computer at a time will attempt to create an IPC structure with a given key value, hence the name. Using a central entity simplifies the design and implementation, but can become a bottleneck in large configurations. Finding a remedy to this problem is left to the time when DIPC is actually running in such configurations.

Users may need to run some programs (e.g. utilities) in all the computers in the system at the same time, and these programs may need to use the same IPC keys. This could create interference. So as to prevent any unwanted interactions, it was decided that distributed IPC structures should be declared by programmers as being so. The programmer just has to specify a flag to do this. The structures are local by default. The mentioned flag is the *only* thing that the programmer should do to create a distributed program. The rest is like ordinary System V IPC programming. Should it not have been the intention to make DIPC compatible with older programs, this system would be totally transparent to programmers.

DIPC Programs

Here, by DIPC programs we mean distributed application programs written to be run using DIPC.

DIPC's programming model is very simple, and quite like using ordinary System V IPC. A Typical scenario is this: a process first creates and initializes the needed IPC structures. After that other processes are started to collaborate on a job. All of them can access the same IPC structures and exchange data. These processes are usually executing in remote machines, but they all could also be running in the same computer, meaning that distributed programs can be written on a single machine and later run in real multi-computers.

One important point to keep in mind about DIPC is that no other UNIX facility is changed to work in a distributed environment. So programmers can not use system calls like fork(), which creates a process in the local computer.

The fact that DIPC programs use numerical keys to be able to transfer data means that they do not need to know where the corresponding IPC structures are. DIPC makes sure that processes find the needed resources just by using the specified keys. The resources could be located in different computers during different runs of a distributed program. This logical addressing of resources makes the programs independent of any physical network characteristics.

Simple techniques allow the mapping from logical computing resources needed by a program to physical resources to be done with no need to re-make the program. As DIPC programs do not need to use any physical network addresses, they do not need recompiling to run in new environments. Of course this does not prevent the programmer from choosing to make his/her program dependent on some physical system characteristics. (S)He could for example hard code a computer address in his code. DIPC programmers are discouraged to do so.

When dipcd is not running, the kernel parts of DIPC are short circuited, causing the system to behave like a normal Linux operating system. So users can easily disable the distributed system. Also, normal Linux kernels are not affected by DIPC programs, meaning that the there is no need to change and recompile these programs when they are to be executed in single computers with no DIPC support.

DIPC’s Distributed Shared Memory

Distributed Shared Memory (DSM) [6] in DIPC uses a multiple-readers / single-writer protocol: DIPC replicates the contents of the shared memory in each computer with reader processes, so they can work in parallel, but there can be only one computer with processes that write to a shared memory. The strict consistency model is used here, meaning that a read will return the most recently written value. It also means that there is no need for the programmer to do any special synchronization activity when accessing a distributed shared memory segment. The most important disadvantage with this scheme is a possible loss of performance in comparison to other DSM consistency models.

DIPC can be configured to provide a segment-based or a page-based DSM. In the first case, DIPC transfers the whole contents of the shared memory from computer to computer, with no regard to whether all that data are to be used or not. This could reduce the data transfer administration time. In the page-based mode, 4KB pages are transferred as needed. This makes multiple parallel writes to different pages possible.

In DIPC each computer is allowed to access the shared memory for at least a configurable time quantum. This lessens the chances of the shared memory being transferred frequently over the network, which could result in very bad performance.

Error Detection in DIPC

DIPC assumes a fail-stop [9] distributed environment. So it uses time-outs to find out about any problem. The at-most-once semantics [1] is used here, meaning that DIPC tries everything only once. In case of an error, it just informs the relevant processes about it, either by a system call return value, or, for shared memory read/writes, via a signal. DIPC itself does not do anything to overcome the problem. The user processes should decide how to deal with the error. This is the normal behavior in many other cases in UNIX .

Security in DIPC

It is important to provide some means to make sure that the data are accessed only by people who are allowed to do so. DIPC uses login names, and not user-ids to identify users. Remote operations are performed after assuming the identity of the person that executed the system call originally. For this to work, the same login name on all the computers in a DIPC cluster should denote the same person.

In order to prevent intrusion to DIPC clusters, addresses of the computers that are allowed to take part in a cluster should be put in a file (/etc/dipc.allow) for DIPC to consult.

Current status of DIPC

DIPC is under development mainly in Iran University of Science and Technology's (IUST) Department of Computer Engineering, but currently people from different parts of the world are working on it. A port to Linux for Motorola 680x0 processors has been completed. This made DIPC a heterogeneous system, as the two versions can talk to each other. DIPC's sources and related documents can be found on the Internet via anonymous FTP at sunsite.unc.edu, in /pub/Linux/system/network/distrib/dipc, or you could download it from DIPC's web page, at http://wallybox.cei.net/dipc.

Conclusion

DIPC is a simple distributed system that works by bringing new functionality to an IPC system designed decades ago. Many of the DIPC's nicer features are the result of its being hidden inside the kernel. Considering its main characteristics, DIPC has the potential to introduce ordinary programmers to distributed programming, thus making Linux one the first operating systems with *usable* and *really used* distributed programming facilities.

There are many experimental distributed systems available for use. Many of them have been implemented in universities, using workstations produced by different manufacturers, and running UNIX variants. The fact that in most cases, researchers did not have free access to the underlying operating system's source code has had a big influence on the design decisions. The availability of source code in Linux has provided new ways to deal with the problems of distributed programming. DIPC is an example of what can be done when one has more access to the operating system sources. Some could mention the problems in porting DIPC to propriety operating systems with no publicly available source code as a draw back. However, in our opinion, propriety operating system vendors and their users are at a loss here, as they can not take advantage of more easy-to-use distributed systems developed by third parties.

The above does not mean that DIPC could not be implemented in other UNIX variants supporting System V IPC, but implies that the port can only be attempted by people with more access to kernel source code.

References

[1] Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall, 1995.
[2] Andrew S. Tanenbaum, *Operating Systems Design and Implementation*, Prentice-Hall, 1987.
[3] Andrew S. Tanenbaum, *Computer Networks*, Prentice-Hall, 1989.
[4] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992.
[5] Robert G. Babb II, editor, *Programming Parallel Processors*, Addison-Wesley, 1988.
[6] Bill Nitzberg and Virginia Lo, "*Distributed Shared Memory: A Survey of Issues and Algorithms*", Computer, August 1991, pp 52-60.
[7] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
[8] PVM web page: http://www.epm.ornl.gov/pvm/pvm_home.html
[9] Alan Burns and Andy Wellings, *Real-Time Systems and Their Programming Languages*, Addison-Wesley, 1990.