DIPC: A Heterogeneous Distributed Programming System

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

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

Michael Schmitz
Laboratory of Chemical Biodynamics
Department of Chemistry
University of California at Berkeley
Berkeley, CA 94720
U.S.A.

E-mail: schmitz@lcbvax.cchem.berkeley.edu

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

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

Abstract

Distributed Inter-Process Communication (DIPC) is a software-only solution to enable people to build and program Multi-Computers. Among other things, DIPC provides the programmers with Transparent Distributed Shared Memory. DIPC is not concerned with the incompatible executable code problem. It also does not change user's data, but DIPC has to understand data produced by its own peer processes in different machines. This paper briefly describes the efforts done to make it a heterogeneous system by enabling it to function under the Linux operating system running on both Intel x86 and Motorola 680x0 processors. Here, different parts of the same application could execute in machines with different architectures

Key Words

Distributed Programming, IPC, Heterogeneous Environment, Distributed Shared Memory, Remote Procedure Call, TCP/IP, Linux.

Introduction to DIPC

DIPC is simple a software-only solution to enable easy and transparent data exchange between the processes of a distributed application [8,9]. Using it, people can build and program multi-computers [1], built of ordinary personal computers connected over a TCP/IP network [6]. The current implementation enables Linux programmers to use UNIX System V IPC mechanisms [2], including shared memories, messages and semaphores, in a distributed environment.

DIPC's services are accessible via the Linux kernel, letting application programmers use the already familiar System V IPC system calls to send and receive data. So, as far as the application programmer is concerned, there are no major changes in DIPC's programming model relative to normal System V IPC programming. There is also no need for any modified compilers or link libraries.

Messages and semaphores are used synchronously, by invoking system calls whenever the program needs them. DIPC uses Remote Procedure Calls (RPC) [3] for their implementation. The shared memories, however, are accessed asynchronously, implying that they can be read or written any time an application programmer wants to. There is no need to do any special synchronization activity before or after the access. DIPC's shared memory uses the single-writer / multiple-readers protocol [7], and employs the strict consistency model [4], meaning that a read will return the most recently written values. Read or write access rights are assigned to computers, and all the processes on that machine can enjoy the rights.

DIPC can manage shared memories in two modes: In the *segment mode*, whenever a process wants to read or write a shared memory whose contents are not present in its computer, the whole contents of a shared memory are transferred between computers, whether they actually need it or not. In *page mode*, 4K byte pages are sent from machine to machine. Our preliminary benchmarks have shown that in certain applications, the performance of DIPC's shared memory in the segment transfer mode can be somewhat better than the page mode. This could be attributed mainly to lower software management overhead. However, because in DIPC there can be only one machine with write access to a shared memory, the page mode allows more concurrent activity in programs that have many processes writing to different pages.

DIPC has two parts: The main part, a program called *dipcd*, runs in the user space with superuser privileges, and a smaller part, hidden inside the kernel, which allows the first part to access and manipulate kernel data. This design is the result of the desire to keep the needed changes in the operating system kernel to a minimum. dipcd creates several processes to do its work. These processes manage all the necessary decision makings and network operations. They use predetermined data packets to communicate with dipcd processes on other computers. All the necessary information for a requested action, including the parameters to a system call, as obtained from inside the kernel, are included in these packets.

DIPC In a Heterogeneous Environment

DIPC was originally developed on Linux for Intel x86 processor family. Right from the early design stages, it was decided that making the system work in a heterogeneous environment should be an important consideration.

DIPC does not care about the incompatibility of different processors' executable programs: The user has to make sure that suitable programs are available in each computer before executing the application. But DIPC and the programmer have to consider the differences in data representation formats. For DIPC to work in a heterogeneous environment, dipcd processes should first understand what is to be done on behalf of a remote peer , and after that, they should use the parameters supplied from the remote kernel in their local machines.

It should be noted that DIPC does not touch data in a distributed shared memory or in the exchanged messages, as doing so would require having some information about the semantics of the application's data usage, which are not obtainable easily. The programmer is responsible for making sure that the data used by the application have the same meaning in all the different computers. Other parameters needed for the operation of any system call are converted automatically by DIPC. For the contents of the messages, there are some techniques to let the programmer do this: (s)he could assign the first byte to hold an identification value. The receiving side could then decide if anything should be done before the data could be used. The same could be attempted for shared memories: The first byte could be set to something representing the architecture that has produced the data. Semaphores could be used to regulate the writes and reads to a shared memory, thus guarantying an orderly access.

The asynchronous nature of the distributed shared memory access in DIPC means that a process can be stopped in the middle of its read or write, and later restarted again, while the shared memory contents have changed. All this could happen without the process noticing anything. This in contrast to the messages and the semaphores, which are used by the program explicitly calling the relevant system calls. This bring up the problem of how to let the process know when there is possible need to convert data in the shared memory .

DIPC tries to address this problem by using asynchronous process notification means: UNIX signals [5]. Two pre-defined signals are sent to processes when they become the reader or writer of a shared memory. Now they can do any necessary action. Unfortunately, the signal mechanism does not allow for any parameters to be passed to the receiving process. This makes it difficult for a process to find out which page of a shared memory should be attended. So it seems necessary for DIPC to use the segment transfer mode when it is used in a heterogeneous environment. The same problem makes it difficult for the programs to use more than on distributed shared memory segment in a heterogeneous environment.

It is clear that dipcd itself also needs to exert some conversions before data produced in one computer can be interpreted and used in another machine. The initial design of DIPC made the receiving side responsible for any conversions required. Each packet would include a pre-defined identifier, specifying the machine architecture that generated the information, in a predetermined field. The receiving side could check it, and knowing the sender architecture, could do any necessary conversions.

It was anticipated that DIPC will be used mostly in homogeneous environments, so the above scheme would, in many cases, make the conversions unnecessary, resulting in more efficiency. The main drawback is that DIPC should know about the data representation formats of all the different architectures on which it was running, making the port to other machines more difficult.

During the port of the system to Linux for Motorola 680x0 processors, another option was explored: Using the TCP/IP network byte ordering convention for the data format reduces the required conversion effort to using the same routines that are part of the TCP/IP networking code. Both sender and receiver side use conversion to and from network byte order, respectively, thus achieving a more general conversion strategy at the expense of additional overhead. This option has been extensively tested during the Linux/m68k port of DIPC.

Both conversion strategies can be used concurrently in the same cluster if all required receive conversion functionality is present in every implementation; special care was taken to ensure that the data conversion scheme may either be chosen at compile time or at startup time. Extension to some strategy for adaptive choice of the conversion scheme is possible, but not implemented yet.

Data conversion is applied to all data fields of the DIPC message headers with the sole exception of the network address of the sending and receiving machine, and to all data fields in the transmitted system call arguments that are relevant on the remote machine. Keeping all data in a format directly understood by the local machine in fact reduced porting DIPC to the 680x0 architecture to supplying the data conversion primitives itself, changing the dipcd code to use the conversion when sending data to the network by adding a wrapper to the network system calls proper, and adding a system call for DIPC’s private use, plus the DIPC shared memory page-fault handler to the architecture-specific parts of the Linux kernel.

Patches to the common Linux IPC kernel code required no modifications, as Linux/m68k is fully integrated into the common Linux kernel source tree with respect to IPC functionality. Ports of DIPC to other architectures should therefore be achieved with minimal effort, making DIPC easily amenable on all supported Linux platforms and raising the issue of integrating DIPC as a general Linux feature.

DIPC’s Development

The port of DIPC to Linux/M68k is nearly complete. The current port makes us confident that DIPC could be ported to Linux for other 32-bit architectures with relative ease. The case of 64-bit systems, however, remains to be tried. Also, implementing most of DIPC in the user space makes us believe that porting it to other UNIX variants supporting System V IPC could be possible..

DIPC’s sources and documents can be obtained and used freely. The latest version of this software can be obtained by anonymous FTP from sunsite.unc.edu, in directory /incoming/Linux, or /pub/Linux/Systems/Networks/distrib/dipc.

Conclusions

Distributed Programming systems seem to be a powerful tool for today’s computing needs, and a heterogeneous one is of much more value. In DIPC’s case, these advantages are supplemented by an easy-to-use programming model.

This work shows that it is possible to design and implement *simple*, and *easy to use* heterogeneous distributed systems. There is a price paid, though: Our preliminary benchmarks have not shown very high remote operation speeds. We are attributing this to the fact that DIPC is mainly a user-space system, with frequent needs to access the kernel data structures, which in a traditional UNIX system, like Linux, means a lot of overhead. However, expecting the speed of computers and communication equipment to rise much more quickly than the number of people trained for distributed programming, we expect this to be of minor significance for DIPC.

References

[1] Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall, 1995.

[2] W. Richard Stevens, *Advanced programming in the UNIX Environment*, Addison-Wesley, 1992.

[3] B. J. Nelson, *Remote Procedure Call*, Ph.D. thesis, Carnegie-Mellon University, 1981.

[4] Bill Nitzberg and Virginia Lo, “*Distributed Shared Memory: A Survey of Issues and Algorithms*”, Computer, 1991.

[5] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[6] Andrew S. Tanenbaum, *Computer Networks*, Prentice-Hall, 1989.

[7] Michael Stumm and Songnian Zhou, “Algorithms Implementing Distributed Shared Memory”, Computer, May 1990.

[8] Kamran Karimi and Mohsen Sharifi, “DIPC: A System Software Solution for Distributed Programming”, To be published.

[9] Kamran Karimi and Mohsen Sharifi, “DIPC: A Distributed Programming System for the Masses”, To be published.