A Brief Survey of High-Level Approaches to Implementing Distributed Applications

Suraj Kurapati
<skurapat@ucsc.edu>
CMPE-185, Winter 2005

6th March 2005

Abstract

Using a low-level approach to implementing inter-process communication for distributed applications burdens the programmer with synchronization issues that are usually irrelevant to the piece of business logic being implemented. This article surveys various high-level approaches which encapsulate inter-process communication whilst enabling the programmer to focus on implementing business logic.

1 Asynchronous Communication

Asynchronous methods of inter-process communication are well suited for distributed applications involving the computation of independent tasks. For example, consider a distributed business application which retrieves the number of new employees hired every year in the last 50 years. Assuming that the process of retrieving the number of new employees hired in a given year does not depend on that of the previous year, we can distribute the overall calculation across 50 different processes—each of which calculates the number of new employees hired during a single year—and combine their individual results into one suitable for the overall calculation.

In addition, asynchronous methods are error-prone when in multi-threaded programming environments [1] due to lack of built-in synchronization facilities, such as semaphores, to prevent communication of obsolete or incorrect data.

1.1 Shared Memory

Before the introduction of message-passing models of asynchronous communication in the late 1970's [2], a technique called "shared memory" was widely in use by supercomputers as a means of inter-process and -processor communication [2]. As the name suggests, this technique involves the reading and writing of data or messages to an area of shared memory. However, the disadvantages of this approach are that (1) it does not scale well for distributed applications running on computing clusters [2] and (2) it has synchronization issues such as race conditions—which further complicate a distributed application because semaphores are necessary to mitigate them. In addition, a shared memory model is quite susceptible to failure because the corruption of the shared memory by a disgruntled process or the loss of electrical power can bring it down, so to speak.

1.1.1 Component Object Model (COM)

COM is a shared-memory model of inter-process communication for the Microsoft Windows operating system [1]. It was originally implemented through the “clipboard” facility and provided naming services through the “registry” facility of said operating system [1]. Though the use of COM is widespread in Microsoft Windows based applications, their programming interface can be quite inhibiting. In particular, (1) “there is no implementation inheritance, thus a component defining a derived interface must implement all functions of the base interfaces again” [1], (2) COM is susceptible to failure because the event of a “registry” corruption [1] can render inter-process communication nonfunctional, and (3) destabilize the remainder of the operating system [1]. In addition, COM functions over a single processor and does not facilitate communication over a network [1, 2].

1.2 Parallel Virtual Machine (PVM)

PVM is a programming interface for distributed applications which can function over a heterogeneous network composed of machines of different architectures and processes implemented in different programming languages [2, 3]. PVM achieves such portability because it provides the necessary “message format transformation to hide differences in computer architectures” [2]. In addition, PVM is “based on the premise that a collection of independent computer systems, interconnected by networks, can be transformed into a coherent, powerful, and cost-effective concurrent computing resource” [3]. In other words, the aim of PVM is to give its user the illusion that her computation is occurring on a single machine [2]. This enables developers to focus on implementing the calculation performed by their distributed application instead of the myriad of complexities introduced by low-level inter-process communication.

PVM is very dynamic, in the sense that processes and machines on the network can be added to and removed from the distributed computation without having to bring it down [2]. It also provides a naming service, which allows processes to dynamically discover other processes and services without being hard-coded to do so [2]. Lastly, PVM is quite fault tolerant as it can dynamically detect and send a notice, indicating which computer became faulty, to functional computers [2]. Alternatively, PVM could command a faulty machine to reboot itself, thereby minimizing the down-time of computational resources in the network.

1.3 Message Passing Interface (MPI)

MPI is a programming interface for distributed applications which was originally developed by supercomputer vendors so that their applications could be compatible with each other [2]. It was designed to function over a homogeneous network of processes and processors, allowing it to take advantage of native network calls to make inter-process communication more efficient [2]. In addition, MPI provides a powerful library of communication procedures that allow point-to-point communication between two processes and point-to-group communication between a single process and a group of processes [2]. However, due to its reliance on network homogeneity, it cannot function over a network of machines with different architectures or processes implemented in different programming languages.

MPI is static, in the sense that processes and machines on the network cannot be added to and removed from a distributed computation without having to bring it down [2]. In addition, it does not provide a common naming service which allows processes and groups of processes to discover each other. Consequently, the allocation of groups and communication paths must be configured before the distributed computation has started. Also, MPI does not have a failure resolution mechanism to revive faulty machines in the computational network [2].

Despite these shortcomings, MPI goes a step further, in terms of message-passing communication methods, in providing support for seamless communication of derived data-types [2, 3]. That is, one is not strictly limited to primitive[1] data-types in inter-process communication.

2 Synchronous Communication

Synchronous methods of inter-process communication are well suited for distributed applications involving the computation of interdependent tasks. For example, consider a distributed business application which calculates a statistical correlation between the number of new employees hired in a given year with that of the previous year, for each year in the last 50 years. In this situation, we cannot simply delegate the computation onto 50 different processes, which perform independently of each other, and combine their results at the end. Instead, each process must communicate with one which is computing the statistical correlation of the year before that of itself. Consequently, synchronous communication, if implemented using low-level methods, become quite complex as the number of dependencies in the functional decomposition of a computation increases.

[1]Data-types integral to a programming language, such as an integer, character, or floating-point number.

2.1 Remote Procedure Call (RPC)

RPC, introduced in 1984 [5], is a programming interface allows a process to execute a procedure or routine on a remote processor as if it was executed on its own processor [1]. It seamlessly encapsulates the synchronous communication necessary to perform such remote procedure calls while also providing support for automatic transmission of procedure-call arguments and return values [1]. However, one can only pass to and receive primitive data-types from RPC [1, 2].

The following sections describe methods of synchronous communication which are based upon RPC.

2.1.1 Distributed Common Object Model (DCOM)

DCOM is a programming interface for the Microsoft Windows operating system [2], which is described by Microsoft as "COM with a long wire" [2] because it adds networking functionality to COM via RPC [2]. Like COM, DCOM utilizes the "registry" facility of said operating system for naming services and is therefore susceptible to failure (See Section 1.1.1). In addition, DCOM can function across a homogeneous network of processes and heterogeneous network of processors—which run the Microsoft Windows operating system [2].

2.1.2 Remote Method Invocation (RMI)

RMI, introduced with the Java Developer's Kit 1.1 [5], is a programming interface specific to the Java programming language. It can be thought of as an object-oriented version of RPC which allows an object in one Java Virtual Machine (JVM) to invoke a method on an object within another JVM—be it local or remote [5, 6]. In particular, RMI facilitates transparent serialization of objects and entire trees of their references—which allows the developer to pass complex (local *and* remote [1]) data-structures as arguments in addition to primitive data-types—and provides a naming service which allows Java objects to discover each other. Also, because the JVM can function on a majority of processor architectures [6, 5], RMI can function over a heterogeneous network of processors and homogeneous network of JVM processes.

Furthermore, RMI changes the way developers think about and design distributed applications [5] by introducing the notion of "stubs" and "skeletons" in decoupling the inter-process communication interfaces[2] and their implementation [5, 6]. The term "stub" refers to the interface seen by a Java application that wishes to invoke a remote procedure, while the term "skeleton" refers to the implementation of the stub's Java programming interface [5, 6]. When a remote procedure is invoked through the stub's interface, the stub communicates with the skeleton in the remote JVM, thereby performing a remote procedure call [5, 6]. In addition, stubs can be downloaded from a remote JVM on demand [1], which makes RMI ideal for dynamic ad-hoc wireless or mobile networks.

[2]The *interface* construct of the Java programming language.

2.1.3 Common Object Request Broker Architecture (CORBA)

CORBA is a programming interface which functions over a heterogeneous network of processes and processors [2] and is "supported by a large industry consortium" [1]. It centralizes inter-process communication through a primary proxy known as the Object Request Broker (ORB) [2], which separates the implementation of computational procedures—known as "object services"[2]—from their RPC interfaces [2, 1]. Like MPI, CORBA is particularly useful for static computational networks, but ill-suited for dynamic ones, such as ad-hoc wireless or mobile networks [1].

3 Graphical Programming

In addition to asynchronous and synchronous programming interfaces for implementing distributed applications, there exist graphical methods which allow one to implement "program decomposition, communication primitives (like PVM and MPI calls) and task assignment to network topologies" [2]. In particular, [2] cites a relatively successful project, which implements the aforementioned goals of graphical programming, named "GRAPNEL". This project also supports an integrated development environment, named "GRADE" [2], which features a distributed debugger, performance monitor, and visualization tools [2]. However, [3] notes that these graphical systems have not had much main-stream acceptance as methods of inter-process communication.

4 Future Research

With the massive transition from low- to high-level methods of inter-process communication in effect during the last twenty years, it would seem that there is a trend in favor of encapsulating complex system-dependent communication routines [2, 1] in standard high-level programming interfaces. With the advent of high-level methods discussed in this article, the creation and management of communication paths between various processes may very well become the next problem—especially in static methods such as MPI and CORBA.

Upcoming graphical programming interfaces seek to facilitate the management of communication paths and resource allocation by allowing one to visually connect processes together. Though they may not have received much attention as of yet [3], graphical programming methods may prove useful in managing computational networks in the future—as the number of machines available to perform distributed computations increases dramatically.

References

[1] F. Mattern and P. Sturm, "From Distributed Systems to Ubiquitous Computing – The State of the Art, Trends, and Prospects of Future Networked

Systems," presented at Kommunikation in Verteilten Systemen (KiVS), Leipzig, Germany, 2003.

[2] P. Kacsuk and F. Vajda, "Network-based Distributed Computing (Metacomputing)," presented at European Research Consortium for Informatics and Mathematics (ERCIM), Computer and Automation Research Institute of the Hungarian Academy of Sciences (MTA SZTAKI), Hungary, 1999.

[3] V. Sunderam, "Heterogeneous network computing: the next generation," Parallel Computing, vol. 23, no. 1–2, Apr. 1997, pp. 121–135.

[4] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," presented at Second USENIX Conference on Object-Oriented Technologies (COOTS), Toronto, Ontario, Canada, June 17–21, 1996.

[5] J. Waldo, "Remote procedure calls and Java Remote Method Invocation," IEEE Concurrency, vol. 6, no. 3, Jul. 1998, pp. 5–7.

[6] Sun Microsystems, Inc. "Java Remote Method Invocation," [Online document], 2003 Dec 11, [cited 6 Feb 2005], Available HTTP: `http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html`