Computer Science Journal of Moldova, vol.5, no.1(13), 1997

Communication Facilities for Distributed Systems

V.Barladeanu

Abstract

The design of physical networks and communication protocols in Distributed Systems can have a direct impact on system efficiency and reliability. This paper tries to identify efficient mechanisms and paradigms for communication in distributed systems.

Judging from the enormous amount of distributed system research carried out over the past decade, information processing experts have come to recognize the advantages these systems possess. These research activities have led to the availability of more than 50 network and distributed systems. However, most of these systems can only partially succeed in attaining the major goals of a distributed system, which include transparency, higher performance, higher reliability and availability, and higher scalability. Of course, attaining all these goals in the first attempt is imposible.

Nonetheless, gradual improvements are posible by learning from existing systems and trying to overcome their limitations.

Distributed transaction-procesing systems must manage such functions as concurency, recovery, and replication. One way to improve their efficiency and reliability is to increase software modularity, which means the separate components should execute in separate address spaces to permit hardware-enforced separation. This structure offers advantages but demands efficient interprocess cimmunication (IPC) service.

The existing communication paradigms are classifyed into three groups: local interprocess communication, remote interprocess communication and communication protocols for both local and wide area networks.

1 Local interprocess communication

To improve local machine performance, Bershad [1] introduced two new mechanisms: lightweight remote procedure call (LRPC) and user-level remote procedure call (URPC). LRPC takes advantage of the control transfer and communiction model of capability — based systems and the adress-space-based protection model of traditional IPC facilities. URPC[2], another cross-address-space communication facility, eliminates the role of the kernel as an IPC intermediary by including commumication and thread management code in each user address space. URPC combines fast cross-address space communication using shared memory with a lightweight threads package that is managed at the user level. The authors significantly expand upon their work on LRPC [3]. In LRPC, the emphasis was on improved performance achieved by exploiting the "common case" of same-machine communication for a multiprocessor operating system kernel. In their previous LRPC work, the authors argued that it is possible to reduce the overhead of a kernel-mediated cross-address space call to nearly the limit possible on a conventional architecture. They observed that the majority of LRPC's overhead was attributed to the fact that the kernel mediates every cross-address space call. In an effort to improve performance and reduce costs, they examine a new system called URPC. In URPC the kernel does not mediate every cross-address space call; rather, management is performed at the user level. In the URPC system, performance is improved by always attempting to schedule another thread from the same address space; the scheduling operation can be handled entirely by the user-level thread code. By preferentially scheduling threads within the same address space, URPC takes advantage of the fact that significantly less overhead is involved in switching a processor to another thread in the same address space than in reallocating a processor to a thread in another address space. Furthermore, the authors suggest that user-level code is the best place to make the decision concerning the context switch itself. The paper defines heavyweight threads, which are threads that make no distinction between a thread, the dynamic component of a program, and its address space, the static component.

They argue that the baggage such as open file descriptors, page tables, and accounting state makes operations on heavyweight threads costly. The authors further argue that kernel-level threads or middleweight threads, in which address spaces and threads are decoupled and the kernel assumes the responsibility for scheduling the thread, do not perform as well as lightweight threads. The paper argues that the kernel's manipulation of middleweight thereads in a general way carries a signification performance penalty. Further, the impact of thread management policy on program performance strongly influences overall throughput of the application: it is unlikely that a kernel-level thread scheduling policy will be efficient for all parallel programs. Therefore, lightweight threads provide a better alternative to these two approaches. In accordance with these observations, the URPC approach emphasizes calls between functions that do not invoke the kernel and that do not unnecessarily reallocate processors between address spaces. These two issues are key to the perfomance enhancements demonstrated in the URPC approach. This paper contains perhaps the most significant results in computer science with respect to the design of operating system software structures for multiprocessor architectures. Its original and thoughtful insights into these issues will have a long-lasting impact on the field.

Both LRPC and URPC were implemented on a DEC SRC Firefly multiprocessor workstation running the Tao operating system [1]. A simple cross-address-space call usig SRC RPC takes 464 microseconds on a single C-VAX processor. LRPC takes 157 microseconds for the same call, and using URPC reduces the call's latency to 93 microseconds.

2 Remote interprocess communication

Several message-based operating systems can reliably send messages to process executing on any host in the network. The **V** system [4] implements address spaces, processes, and the interprocesses communication protocol in the kernel to provide a high-performance message-passing facility. All high-level system services are implemented outside the kernel in separate processes. **Mash** uses virtual-memory techniques to optimize local IPC. Remote comminication goes through a user-level server process, which adds extra overhead. **Amoeba** uses capabilities for acces control and message address. It has a small kernel, and most features are in user processes.

However, not all the systems use the small-kernel approach with remote IPC outside the kernel. In Sprite [5], the IPC is through a pseudodevice mechanism. Sprite kernel communication is through Sprite kernel-to-kernel RPC. RPC in x-kernel [6] is also implemented at the kernel level. Table 1 shows the performance of various RPCs over Ethernet.

Table 1. Performance data for remote procedure calls. (The Sun 3/75 is a 2-MIPS machine, and the Sun 3/60 is a 3-MIPS machine)

System	**RPC Type**	**Architecture**	**Latency**	**Latency by MIPS**
V	User level	Sun 3/75	2.50 ms	5.0 ms
Mash	User level	Sun 3/60	11.0 ms	33.0 ms
Amoeba	User level	Sun 3/60	1.1 ms	3.3 ms
Sprite	Kernel level	Sun 3/75	2.45 ms	4.9 ms
Sprite	Kernel level	Sun 3/75	1.70 ms	3.4 ms

3 Comunication protocols

Communication protocols provide a standard way to communicate between hosts conected by a network. Datagram protocol such as IP are inexpensive but unreliable. However, more reliable protocols, such as virtual-circuit and requiest-response protocols, can be built on top of datagram protocols.

The versatile message transaction protocol (VMTP) is a transport-level protocol that supports the intrasystem model of distributed processing [4]. Page-level of file access, remote procedure calls, real-time datagrams, and multicasting dominate the communication activities. VMTP provides two facilities, stable addressing and message transactions, useful for implementing conversions at higher levels. A stable address can be used in multiple message transaction as long as it remains valid. A message transaction is a reliable request-response interaction between addressable network entities (ports, processes, procedure invocations). Multicasting, datagrams, and forwarding services are provided as variants of the message transaction mechanism.

Using virtual protocols and layered protocols, the x-kernel implements general-purpose yet efficient RPCs. Virtual protocols are demultiplexers that route the messages to appropiate lower level protocols. For example, in an Internet environment, a virtual protocol will bypass the Internet Protocol for messages originating and ending in the same network. The support of atomic broadcasting and failure detection within the communication subsystem simplifies transaction processing software and optimizes network broadcasting capabilities [7].

4 Forms of interprocess communication

OS/390 OpenEdition services provide three special ways for programming processes to communicate:

Message queues, Semaphores, Shared memory

These forms of interprocess communication extend the possibilities provided by the simpler forms of communication: pipes, named pipes or FIFOs, signals, and sockets. Like these forms, message queues, semaphores, and shared memory are used for communication between processes. (Sockets are the most common form of interprocess communication across different systems.)

a) Message queues

OS/390 OpenEdition services provide a set of C functions that allow processes to communicate through one or more message queues in an operating system's kernel. A process can create, read from, or write to a message queue. Each message is identified with a "type" number, a length value, and data (if the length is greater than 0).

A message can be read from a queue based on its type rather than on its order of arrival. Multiple processes can share the same queue. For example, a server process can handle messages from a number of client processes and associate a particular message type with a particular client process. Or the message type can be used to assign a priority in which a message should be dequeued and handled.

b) Semaphores

Semaphores, unlike message queues and pipes, are not used for exchanging data, but as a means of synchronizing operations among processes. A semaphore value is stored in the kernel and then set, read, and reset by sharing processes according to some defined scheme. A semaphore can have a single value or a set of values; each value can be binary (0 or 1) or a larger value, depending on the implementation. For each value in a set, the kernel keeps track of the process ID that did the last operation on that value, the number of processes waiting for the value to increase, and the number of processes waiting for the value to become 0.

A semaphore is created or an existing one is located with the semget() function. Typical uses include resource counting, file locking, and the serialization of shared memory.

c) Shared memory

Shared memory provides an efficient way for multiple processes to share data (for example, control information that all processes require access to). Commonly, the processes use semaphores to take turns getting access to the shared memory. For example, a server process can use a semaphore to lock a shared memory area, then update the area with new information, use a semaphore to unlock the shared memory area, and then notify sharing processes. Each client process sharing the information can then use a semaphore to lock the area, read it, and then unlock it again for access by other sharing processes.

5 Communication Transport Protocols

The communication transport protocols will be exemplified using such sistems as *Isis* and *Raid*.

Isis is a Toolkit for building distributed applications. Isis was developed as a UNIX-oriented distributed programming environment, although the system has now migrated to a number of other platforms.

Isis implements several protocol layers, using an architecture that is stacked somewhat like the ones seen in TCP/IP or the OSI architecture. The highest layer of protocols, labeled **vsync**, is concerned with supporting the full virtual synchrony model for multiple groups. Its overheads are as follows:

- It adds ordering information to messages, using, called **compressed vector timestamps**. Timestamp size grows in proportion to the number of processes permitted to send in a process group, which is generally between 1 and 3 in Isis applications, but can rise to 32 in certain types of parallel codes.

- It may delay messages for atomicity and ordering reasons. There are a number of possible delay conditions. For example, transmission of a message may have to be delayed until some other message has been sent, a message can arrive before some other message should precede it, the delivery of a totally ordered multicast may need to be delayed until the delivery ordering is known, and so forth.

- When a process group membership change is occurring, this layer ensures that each message is delivered atomically, before or after the membership change, at all members.

Below the virtual synchrony layer is a multicast transport layer. This layer has responsibility for delivering messages in the order they were sent (point-to-point ordering only), without loss or duplication unless the sender fails. Callbacks to the virtual synchrony layer report on successful delivery of a message to its remote destination. The multicast transport layer exploits several message transport protocols:

- The UDP transport layer supports point-to-point communication channels using the UDP protocol (user datagram protocol). This layer implements reliability using a windowed acknowledgement scheme, much as TCP does over the IP protocol. If process p has channels to process q and r, notice that the actual UDP packets travelling over these respective channels may be very different. Isis supports various point-to-point and RPC communication mechanisms, and q and r may not be members of the same set of process group. Since the packing algorithm will be applied to the aggregate traffic from p to q and r, the packet stream used in each case will differ.

- The IP-multicast transport layer. IP-multicast is similar to UDP, in that it provides an unreliable datagram mechanism. However, whereas UDP operates on a point-to-point basis, IP-multicast supports group destination and the associated routing facilities. IP-multicast is not reliable, hence protocol implements its own flow-control and error correction logic. Notice that although IP-multicast offers a reduction in the number of packets needed to send a given message to a set of a destinations, this assumes that identical packets must be sent to the destination processes.

- A transport layer using shared memory for local communication. This layer is planned for the future: Isis does not currently optimize for local communication, although the growing availability of shared memory in modern operating systems makes this feasible. An extension of Isis to communicate between processes on the same machine using a shared memory pool would greatly enhance local performance, as well as remote performance for groups having some local destination.

- A transport layer for Mach IPC. This layer is experimental.

The above layering is used when application programs are able to communicate directly. Most Isis communication is direct (called "by-pass" communication), and although there are other communication paths which are not described here.

In Raid (Raid has been developd at Purdure University on Sun workstation under the Unix operating system in a local area network), each major logical component is implememted as a server, which is a process in a separate address space. Servers interact with other processes through a high-level communication subsitem. Currently, Raid has six servers for transaction management: the user interface (UI), the action driver (AD), the access manager (AM), the atomicity controller (AC), the concurrency controller (CC), and the replication controller (RC). High-level name service is provided by a separate server, the *oracle*.

Raid's communication software is called Radicomm. The first version, Radicomm V.1, was developed in 1986. Implemented on top of SunOS socket-based IPC mechanism using UDP/IP (User Datagram Protocol/Internet Protocol), it provides a clean, location-independent interface between the servers. Radicomm V.2 was developed in 1990 to provide milticasting support for the AC and RC services. Radicomm V.3 was developed to support transmission of complex database objects. It is based on the explicit control-passing mechanism and shared memory.

Summary

The key difference between a centralized operating system and a distributed one is the importance of communication in the latter. Various approaches to communication in distributed system have been proposed and implemented. Wide-area distributed systems, connection-oriented layered protocols suuch as OSI and TCP/IP are sometimes used because the main problem to be overcome is how to transport the bits over poor physical lines.

For LAN-based distributed system, layered protocols are rarely used. Instead, a much simpler model is usually adopted, in which the client sends a message to the server and the server sends back a replay to the client. By eliminating most of the layers, much higher performance can be achieved. Many of the design issues in these message-passing systems concern the communication primitives: blocking versus nonblocking, buffered versus unbuffered, reliable versus unreliable, and so on.

The problem with the basic client-server model is that conceptually interprocess communication handled as I/O. To prevent a better abstraction, the remote procedure call is widly used. With RPC, a client running on one machine calls a procedure running on another machine. The runtime system, embodied in stub procedures, handles collecting parameters, building messages, and the interface with the kernel to actually move the bits.

Although RPC is a step forward above raw message passing, it has its own problems. The correct server has to be located. Pointers and complex data structures are hard to pass. Global variables are difficult to use. The exact semantics of RPC are tricky because clients and servers can fail independently of one another. Finally, implementing RPC efficiently is not straightforward and requies careful thought.

RPC is limited to those situations where a single client wants to talk to a single server. When a collection of processes, for example, replicated file servers, need to communicate with each other as a group, something else is needed. Systems such as Isis provide a new abstraction for this purpose: group communication. Isis offers a variety of primitives, the most important of which is CBCAST. CBCAST offers weakened communication semantics based on causality and implemented by including sequence number vectors in each message to allow the receiver to see whether the message should be delivered immediately or delayed until some prior messages have arrived.

References

[1] B.N.Bershad. High-Performance Cross-Address Space Communication, PhD thesis, Tech. Report 90-06-02, University of Washington, Seattle, 1990.

[2] Bershad Brian N., Anderson Thomas E., Lazowska Edward D. User-level interprocess communication for shared memory multi-

processors, ACM Transactions on Computer Systems vol.9, No.2 (May 1991), pp.175–198.

[3] Bershad B.N., Anderson T.E., Lazowska E.D. and Levy H.M. Lightweight remote procedure call. ACM Trans. Comput. Syst. (Feb. 1990), 37–55.

[4] D.R.Cheriton. The V Distributed System, Comm. ACM, Vol.31, No.3, Mar. 1988, pp.314–333.

[5] J.K.Ousterhout. The Sprite Network Operating System, Computer, Vol.21, No.2, Feb. 1988, pp.23–36.

[6] L.Peterson. The x-kernel: A Platform for Accessing Internet Resources, Computer, Vol.23, No.5, May 1990, pp.23–33.

[7] E.Mafla and B.Bhargava. Implementation and Performance of a Communication Facility for Distributed Transaction Processing, Usenix Symp. Experiences with Distributed and Multiprocessor Systems, Usenix Assoc., Berkeley, Calif., Mar. 1991, pp.69–85.

Vladimir Barladeanu,
Department of Computer Science
Politechnical University of Bucharest
Splaiul Independentei 313,
Sector 6, Bucuresti, Romania.
Phone: (40-1) 410-04-00, (40-1) 311-16-76
E-mail: *vladb@disco.cs.pub.ro*
vladb@arexim.ro

21 October 1996