Remote Procedure Calls In The Distributed Enterprise

© 1993, T.A. Daneliuk
DePaul University
csctad@hawk.depaul.edu

November 20, 1993

Abstract

There is a great deal of interest currently in how the large computing enterprise is to be integrated into a "seamless whole". Organizations find themselves with a wide variety of hardware, operating systems, and networks in place to run their mission-critical applications. Remote Procedure Call (RPC) has received a great deal of attention as the candidate integration mechanism. Here we examine the structural problems to be solved in the large enterprise and comment on the viability of RPC as such an integration mechanism.

1 Introduction

The advent of low-cost desktop computing has forever changed the shape of the large computing enterprise. [1] Where once centralized mainframe computers provided the only viable solution for such an enterprise, today more and more of the work is being done on the user's desk. As the economics have moved computation away from a single, central site, a *de facto* distributed computing environment has been created with most of the mission-critical data still in a central site, but computation spread throughout the entire enterprise. Moreover, because this distributed system was largely not planned, but evolved over time, it is a rat's nest of incompatible operating systems, network protocols, data formats, and applications. This fact alone makes managing the now distributed enterprise in a single, coherent fashion virtually impossible. Worse yet, such systems are largely incapable of sharing data or doing any sort of cooperative computing. In short, the economics of the new technology has created a legacy of "islands of computing", each island invisible to the other.

It is clearly desirable that such an enterprise be integrated

[1]The term "large enterprise" is a bit vague and is used loosely in the popular literature. However, we use the term here to describe an organization which has both mainframe and desktop computing (and possibly mid-range computers as well), uses both Local Area and Wide Area Networks, serves a large number of users (5000 or more), and requires continuous uptime and availability for its mission-critical applications. This definition is typified by large corporate information systems but is less representative of large academic or research systems. For instance, the Internet would not qualify because, although continuous network availability is desirable, it is not required - i.e., There is no sense of "mission criticality" in day-to-day operations.

into one seamless whole. Not only can applications work together more effectively in such an environment, but system management, code reusability, and a general reduction in the long-term cost of ownership are facilitated by an integrated enterprise. Hence, a great deal of effort is today being expended to find a technically feasible and economically realistic means to unify the large enterprise. [2]

2 Application Integration Models

Any such enterprise integration has several elements to it. Certainly, a means of normalizing application-to-application semantics is needed. Similarly, there is a need to provide a common format for the exchange of data. But, both these problems pale into insignificance when compared to the problem of conquering the disparities between networking infrastructures.

Historically, networking technologies and protocols were defined by two entities, private industry and government. [3] Each source had different motives and hence produced radically different networking technologies. Industry needed highly reliable and available wide-area networks which could secured

[2]It is worth noting that the goal is more precisely stated as *providing the image of a unified system from the point-of-view of a mission-critical application.* That is, applications should "see" a unified enterprise even though the underlying system is geographically distributed and composed of heterogeneous components.

[3]This is certainly true for the United States, in any case. In Europe and other industrialized nations there was also an influence from the national PTTs and standards bodies such as OSI and CCITT.

and managed from a central site. So, for instance, IBM's LU 6.2 SNA product is largely configured statically in a central location. By contrast, government wanted communications products which simplified connectivity between geographically separated sites without the need for centralized administration. This bias can be seen, for example, by the autonomy given to local name servers in the Domain Name Service (DNS) which is most often associated with TCP/IP and the Internet. It is especially noteworthy that the commercial networking technologies have largely been built with the idea of connecting *the many to the few* (as in terminals to hosts) whereas research and academic networking strongly prefers to connect *the many to the many* (as in hosts to hosts). [4]

The essence of integrating the large enterprise thus becomes first the problem of conquering the semantic disparity between various networking technologies. It is also widely accepted that any such mechanism should alleviate the applications programmer from having to know the intricacies of the various protocol stacks and networking technologies in use throughout the enterprise. It is thus useful to examine the major models which have been proposed which "insulate" applications from the network and integrate applications across the network into a unified whole.

[4]This will figure prominently in a later discussion dealing with protocol session loads in a distributed enterprise.

2.1 Protocol Conversion

Among the earliest attempts to conquer networking heterogeneity was *Protocol Conversion.* In this model, translation software (and possibly hardware) is deployed throughout the enterprise to allow translation between any pair of protocols in use. As traffic between applications flows through the enterprise, it is transparently converted from protocol to protocol.

There are two primary drawbacks to this model. First of all, it is usually the case that there is not a one-to-one mapping of semantics and services between various networking protocols. This becomes a problem particularly when translating from a richly functional protocol to a simpler protocol. Consider, for example, the case of mapping a TCP/IP broadcast onto a protocol which does not support broadcasting. The protocol converter is forced to mimic the broadcast by sending individual messages to each of the end points in the domain of the more primitive protocol. This is so inefficient as to be practically worthless.

Protocol conversion also has a severe limitation because of the complexity that must be managed. If a given enterprise uses n different protocols, it will need to provide $n(n-1)$ different protocol conversions. In other words, given n different protocols, there is $O(n^2)$ conversion complexity. As n is increased, the amount of work necessary to support the new protocol grows rapidly. For example, if the enterprise is using only two protocols and adds a third, the number of protocol conversions possible rises from 2 to 6. However, if an enterprise is using four protocols and adds one more, the number of possible protocol conversions rises from 12 to 20! Again, this is, for all intents and purposes, impractical in a large environment.

2.2 Migration To Homogeneity

In this model, the enterprise recognizes the inherent complexity of perpetuating multiple networking technologies. In response, a plan is implemented which migrates the entire enterprise to a common networking platform over time. This model is most often offered by the advocates of so-called *Open Systems*. Here, we are told, the solution is to simply migrate everything to *the* open protocol (TCP/IP or OSI, depending on which branch of theology one was schooled in).

The problems with this approach are legion, but several present themselves immediately. First, a large enterprise will almost certainly not be able to do everything needed with a single protocol. [5] For instance, TCP/IP cannot be deployed effectively as a wide-area protocol where the enterprise is built around a single, centralized data center. Even the largest mainframe front-end processors cannot provide the resources necessary to support tens of thousands of TCP sessions that are connecting to that single node on the network. [6]

[5]The most obvious supporting argument here is that new protocols evolved precisely to solve problems which their predecessors could not!

[6]This is not an exaggeration of session load. By the definition of a "large" enterprise

A second objection to this sort of enforced homogeneity is that the protocol of choice may not be available or well-implemented on all the different platforms in use by the enterprise. For instance, TCP/IP is just beginning to mature on IBM MVS systems. LU 6.2, while implemented on selected Unix systems, is in many cases too unstable or incomplete to be used as the single protocol of choice.

Thirdly, older applications which run an enterprise typically have network-specific dependencies embedded in them. In order to migrate to a single new networking technology, these applications must be modified or even completely rewritten. This is often cost-prohibitive. Even in those cases where such application retooling is economically viable, the time it takes to migrate many mission-critical applications may not be reasonable.

Finally, retooling the networking infrastructure of a large enterprise is not economically feasible. Most such organizations have millions of dollars of investment in their networks. Even if technically feasible, any proposed migration plan must be economically rational. Given the investment already in place in the enterprise, this is frequently difficult to do.

---

given previously, the smallest system under consideration here will have 5000 users. At a minimum, each of these users will have a single active process, typically a terminal emulation like Telnet or TN3270. If each of these users on average runs only one other application, say email or file transfer, the total load at the central host will be 10,000 sessions. This is actually a rather small example. Typical enterprises will have on the order of 20,000 users with a corresponding load of 40,000 sessions at the central host. No TCP/IP implementation today is capable of supporting this sort of overhead.

2.3 Middleware

An increasingly popular technique for conquering network diversity is that of *Middleware*. Middleware broadly refers to software which provides the illusion of network homogeneity to an application while at the same time actually operating over a variety of networking technologies. Examples of middleware include messaging systems and distributed queuing managers. Here, a homogeneous programming paradigm is presented to all applications regardless of where they reside. The middleware software then translates the semantics of that paradigm into specific calls meaningful to the underlying network services.

While middleware is quite effective in this regard, it poses a number of problems which must be overcome. First, the addition of middleware also adds a new virtual address space which must be managed. For instance, a distributed queuing manager will have a set of queue names which must be associated with underlying network addresses. Secondly, adding a constant layer of software between every application and the underlying network can have adverse performance impact. Happily, these problems seem to be diminishing as middleware vendors produce successive generations of their products.

2.4 Programming Language Extensions

Remote Procedure Calls fall into the category of *Programming Language Extensions.* [7] Here, the applications programmer need not be explicitly aware of the mechanics of networking. Rather, familiar constructs in the programming language of choice are used. These are then mapped to appropriate networking facilities as needed by the underlying language runtime and operating system services. This model is popular precisely because it requires a minimum of training to use effectively. Since the “extension” of the programming language is actually done by the language runtime and/or the operating system, programmers remain largely unaware of the role of networking in enabling the application and proceed to write applications as they always have.

Today there is an enormous interest in RPC. In fact, it is fair to say that the only mechanisms of enterprise integration under serious consideration are RPC and middleware. The “open systems” vendors have firmly aligned themselves with RPC. Both SUN Microsystems Open Network Computing (ONC and ONC+) and the Open Software Foundation Distributed Computing Environment (DCE) are built on an RPC foundation. [8]

[7] It is also common to see RPC categorized as middleware. The distinctions between middleware and programming language extensions are somewhat arbitrary and either categorization (or both) for RPC is probably appropriate.

[8] The middleware vendors, who largely vend proprietary products, are responding with their own consortia such as the Message Oriented Middleware group. These vendors have a strong case for their products because, although both SUN and OSF claim to be “open”, their respective RPC products cannot interoperate with each other!

Because of this very visible support for RPC, some have suggested it as *the* mechanism of enterprise integration. It is thus useful to examine RPC in more detail and evaluate its efficacy in integrating the large enterprise.

3 Overview Of RPC

A *Remote Procedure Call* is exactly what it's name suggests. Namely, the programmer invokes a function call with the possibility (depending on how the underlying system has been configured) that the function will actually execute on a remote server. The key idea here is that the programmer sees the normal call-return semantics of the programming language in use. The program has no explicit notion of what is executing locally or remotely. For all intents and purposes, RPCs act like slow function calls.

> The remote procedure call paradigm for programming focuses on the application. It allows a programmer to concentrate on devising a conventional program that solves the problem before attempting to divide the program into pieces that operate on multiple computers...
>
> The remote procedure call model uses the same procedural abstraction as a conventional program, but allows a procedure call to span the boundary be-

> tween two computers. [9]

The complexity of managing the details of the required networking calls is "pushed down" to the RPC software so that the applications programmer is insulated from them. In effect, the basic unit of addressability is a function name, not a network address. As the topology or other operational details of the network change, the RPC configuration must be changed, but not each and every application making use of it.

Because RPCs are expected to run in heterogeneous hardware environments, there is typically some form of data normalization scheme in each different RPC implementation. This allows the caller to declare a piece of passed data to be, say, `integer`, and insures that the called function will understand it as such. This is quite similar to the case where separately compiled modules which are later linked together must agree on parameter order and size for proper operation. SUN, for example, has a data exchange standard called *eXternal Data Representation* or XDR.

> The XDR standard has two aspects. The first addresses the issue of data representation; it defines a uniform way to represent data types. The second addresses data description; it defines a language that can describe data structures of arbitrary complexity in a standard way. [10]

[9] [CS93, Page 235]

[10] [Cor90, Page 12]

Underlying the call-return interface presented to the applications programmer, is the RPC mechanism itself. When a function is called which will ultimately run on a remote machine, the caller is actually invoking the so-called “client stub” on it’s own machine. This is a piece of RPC logic which uses the underlying network protocol stack to deliver the call and its normalized parameters to the remote machine. This information is delivered to the “server stub” on the remote machine which actually invokes the function in question. When the function completes the call, the return value is sent from the server stub to the client stub, again using the underlying networking protocols and services. Finally, the returned value is passed from the client stub back to the original calling software. All of this is accomplished invisibly to the original calling application, which is just waiting for the function call to complete before continuing.

RPCs are frequently cited as the natural way to achieve *client-server* computing.

> The remote procedure call can therefore be used to realize a model popularly known as the ‘client-server model’, in which the client requests a server to perform a task of which it is capable and waits until the server completes the task and returns the required result.
>
> The client-server model splits the information management function of the operating system into two components: a front end (client), where the data is manipulated and displayed; and a back end (server),

where data is stored and accessed. The users interact only with the client directly through a program, and not with the server. [11]

It is, in fact, exactly this model of client-server computing as enabled by RPC that is today the proposed mechanism for creating an integrated enterprise.

4 Critique' Of RPC

In evaluating RPC as a mechanism for enterprise integration, it is important to distinguish RPC as a *paradigm* from RPC as an *implementation.* Most of the features or limitations discussed in the following sections are not inherent to the concept of remote procedure calls *per se* . Rather, they are artifacts of RPC as currently implemented.

In the same spirit, this analysis limits itself to features found in the only two RPC implementations which have current marketplace viability: SUN's ONC/ONC+ product, and The Open Software Foundation's DCE proposal. [12]

[11][KM91, Page 5]

[12]In some sense, this is a bit generous. The only RPC actually implemented and in use extensively today is SUN's. DCE, while discussed widely, is realistically only in experimental implementation at this time. Nevertheless, DCE has a significant "paper" following. A large number of vendors have pledged to implement and deploy it "in the near future".

4.1 Benefits Of RPC

RPC clearly achieves the goal of insulating programmers from the underlying details of the network. The programmer sees nothing more than the familiar call-return semantics of the language in use. [13] Beyond merely simplifying the programming process, there is significant benefit in preserving the skill of the programming staff. To write RPC-based applications, a programmer requires no large amount of incremental training.

Perhaps more significantly, different distribution schemes can be tried quickly without major programming impact when RPC is used. The basic unit of "addressability" from the perspective of an RPC-aware program is the name of a function. Such an application has no direct knowledge of native protocol addresses or networking semantics. When a new distribution of workload is to be implemented, the functions to be remotely executed are ported to the new server locations. The appropriate RPC configurations throughout the network are then modified to reflect the new locations of service. So, for example, a program will still call function `foo()` without being aware that `foo()` now runs in a different place. Here, the greatest benefit is that new applications can be developed as relatively small pilot projects. Once the application is stable and needs to grow in computational capacity, it can be distributed across more capable servers with minor changes to the overall system.

[13] There is, of course, a great deal of RPC configuration work "under the covers" which a systems administrator must undertake. This is quite reasonable because each and every application is not burdened with such detail.

Finally, RPC systems tend to do a good job of typing and normalizing data. This further insulates applications software from the vagaries of the underlying network and CPU architectures. As these enabling technologies inevitably change, the core logic of enterprise applications remains unaffected.

4.2 Limitations Of RPC

At first glance it would seem that RPC is an intuitive and obvious way to integrate large enterprises. A deeper analysis, however, exposes some limitations which bear heavily on this technology's viability in such environments.

4.2.1 Synchronous Semantics

The entire paradigm of RPC revolves around the idea of a function call. While this is a simple, well understood kind of program behavior, it is often inappropriate for building large distributed systems. When a function is called, the calling code halts until the function returns. This "spin locking" or "blocking" semantic is generally unacceptable in large environments for several reasons.

First, this kind of synchronous operation generally yields poor overall performance because the calling routine must wait for the called routine. Consider the case where the caller happens to be a server process. Overall server performance will be very low if the server must wait as each RPC runs to completion. Obviously, the server needs to continue processing other tasks while waiting for RPC completions if it is to be maximally efficient.

Synchronous semantics also pose a problem because they implicitly assume that both the calling and called functions will be simultaneously available and sane. It is frequently the case in the distributed enterprise that the producer of data and the consumer of data need to be independent of each other in time. This is because applications are typically crafted and distributed to reflect a business or organizational model rather than the networking model. If a data producer cannot reach its corresponding consumer, data production should continue nonetheless. Applications built strictly on the call-return mechanism of RPC will fail if, say, a call to the (remote) consumer cannot complete because a network link is down.

Another problem with RPC's synchronicity arises when one considers the problem of *Mutual Exclusion* in a distributed environment. One can view a distributed system as a loosely-coupled parallel processor with the network replacing a local machine bus. In such a model it is important to guarantee exclusive access to certain common resources such as shared data, synchronization flags, and so on. Some attempt has been made to support this directly in the RPC semantics:

> The SUN RPC mechanism specifies that at most one remote procedure in a remote program can be invoked at a given time. Thus, RPC provides auto-

matic mutual exclusion among procedures within a given remote program. [14]

Here, the cure may be worse than the disease. We get mutual exclusion by enforced serialization at all times. Once again, synchronicity (as a serialization mechanism) can cause a significant loss of parallelism and thus, performance.

A final problem with the synchronous call-return semantics of RPC is that there is no tail-recursion elimination in the distributed call tree. If an initial RPC calls another which in turn calls another RPC and so on, each called function can only return to its caller. This pairwise call-return scheme means that performance is sacrificed when multi-server RPCs are needed to satisfy an initial client request. There is no way to have the RPC which was invoked last return *directly* to the original caller. In large environments it is quite common to need the processing of several servers before a given client's request can be satisfied. [15] RPC semantics guarantee that such situations will run more slowly than they really have to.

A number of solutions have been proposed to overcome the

[14][CS93, Page 243]

[15]In fact, an increasingly popular work distribution scheme is to create *functionally split* processing topologies wherein each hardware node does one, and only one task. This gives the systems designer very fine-grained control of the distributed environment. Hardware can be chosen to enable maximum performance and minimize cost for a particular function. For example, parallel processors can be used to execute scheduling problems (which are usually NP-Complete) while high I/O bandwidth machines can be used for database management. By definition, such an approach requires the work of multiple servers to satisfy a complex client request.

fundamental concern of RPC synchronicity. One possible solution is to run RPCs asynchronously - i.e., Issue an RPC call and then go on processing with the expectation of collecting the results later. SUN RPC, for instance, does support some notion of *Nonblocking RPC.* It is presented as essentially a one-way message passing scheme. The client invokes an RPC which returns immediately because no reply is expected. Nonblocking RPC does alleviate the problems of synchronicity but at a rather high cost:

> Another consequence that Nonblocking RPC has on the semantics of the request is that the RPC Library no longer takes responsibility for retrying a request for which a reply has not been received. This means that if the RPC is sent over an unreliable datagram transport such as UDP, the message could be lost without the client's knowing it. Applications using Nonblocking RPC must be prepared to handle this situation. [16]

In effect, Nonblocking RPC gives the programmer asynchronous messaging semantics at the expense of reliability and program simplicity.

The more general problem with Nonblocking RPC is that it does not correlate client requests and server replies. This can be done by either explicitly polling the server or using a SUN

[16] [Cor90, Page 148]

rendezvous mechanism known as *Callback RPC*. In either case, however, the burden of correlating asynchronous client requests and server replies lies with the application. A preferable model would be to have the underlying system do this correlation and free the application to inspect the results only as needed.

Another possibility to overcome this limitation is to run *Threaded RPCs*. Here multiple execution paths, or "threads" run simultaneously within one process context. The idea is that, while a given thread may block waiting for an RPC to complete, the overall process (i.e., other threads) can continue running. The DCE design calls for its RPC implementation to be thread-aware:

> To improve program performance, RPC server stub routines use threads internally, enabling RPC-based servers to handle multiple client requests simultaneously. Threads can also be useful when writing advanced client applications. [17]

Threading is probably the most realistic approach to address some of the problems associated with RPC's synchronized semantics. (Note that threading does not alleviate either the producer/consumer or tail-recursion problems outlined above.) However, a large enterprise is likely to have a variety of operating systems, not all of which will support threading. Even in environments where threads are "supported" one must be careful. Often, all that is actually present is the POSIX `p-threads`

[17][RKF93, Page 55]

library interface and threads are merely emulated in the operating system's user space. In this sort of system, *when one thread blocks all other threads in that process space also block.* This reduces the so-called threaded environment to exactly the same behavior of a system which has only heavyweight processes. i.e., An RPC running under such a thread implementation will be completely synchronous, exactly as if no threads were present.

4.2.2 Protocol Dependencies

For all practical purposes, RPCs today operate in a single protocol environment exclusively (TCP/IP). This alone guarantees that RPC cannot be used as an enterprise integration vehicle. IBM's SNA networking products are ubiquitous in the large enterprise and must be accommodated. Furthermore, the large installed base of LANs demands support for protocols like IBM NetBIOS and Novell IPX/SPX as well.

SUN does have a *Transport Independent RPC* product [18] which is claimed to be free of protocol-dependencies. However, this product is not available widely outside of UNIX SVR4. Even if it were, Transport Independent RPC does not transparently move traffic across multiple underlying protocols.

The transport selection mechanism provides a way
to choose the transport to be used by the application.

[18][Cor90, Page 251 ff]

> Each transport belongs to a class, such as datagram or circuit-oriented. You can specify either a specific transport required for the application or a particular class of transport, or you could leave the decision up to the user at runtime. [19]

The constraint here is that you must choose *a single* transport protocol. There is no way to initiate an RPC on, say, a TCP/IP-connected node and have the requested function execute on an LU 6.2-connected machine. So, even with Transport Independent RPC, an application can only engage one protocol per RPC.

4.2.3 Session Loading

There is little question that building robust distributed systems requires *reliable* application-to-application dialog. In the case of RPC this means that the underlying delivery protocol must be connection-oriented. In fact, each simultaneously active RPC requires its own distinct connection. As discussed previously, the existing enterprise networks were built with centralized computing in mind. It is practically impossible to terminate the required number of connections (transport sessions) at a single site on the network.

Even assuming that the total session load could be supported, a large network with many active sessions is inherently expensive

[19][ibid, Pages 251-252]

and hard to operate. It is expensive because a large session load implies that more computer resources (memory and CPU) are needed to maintain the state of all the active connections. It is hard to operate because, when the network fails, it is very time-consuming to reestablish the thousands of needed connections all at once. Such network recovery operations can literally take *hours* which is unacceptable in most enterprise operations.

4.2.4 Legacy Integration

Any enterprise integration strategy must consider the large base of existing applications. More particularly, such a strategy should provide for the integration of these software *legacies* with the newer applications and technologies. The bulk of legacy applications were written with the old host-terminal paradigm in mind. The "currency of the realm" for such applications is not a function call but a *screen image*. That is, older applications expect to get work requests in the form of a formatted screen and return results in the same format.

Many older applications were explicitly written to be run under a transaction monitor like IBM CICS. Such *TP Applications* are event driven. The user creates what amounts to a message and ships it to the TP monitor by pressing the transmit key on their terminal. The TP monitor then starts up the relevant applications code based on the content of that message. Mating such an application to the larger enterprise via function calls can be quite clumsy. A more natural model is to interact with legacy TP applications via messaging middleware.

4.2.5 Enterprise Management

In the large distributed environment there is an increasing emphasis on remote management. The idea is to distribute computation and possibly data, *but manage centrally.* As currently implemented, there is no facility for centralized management of the underlying RPC configurations scattered across the network. This is a distinct drawback. RPC is able to offer the programmer as simple interface at the expense of substantial and complex underlying RPC configuration. It is impractical to insist that such configurations must be manually managed on a site-by-site basis.

It could be argued that, since there is a network connecting all machines participating in RPCs, the systems administrator could simply remotely login to each machine to administer its RPC configuration. Not only is this manual approach ineffective in a large environment, the desire exists to do such administration programmatically, not interactively. This is why network management protocols such as SNMP and CMIP have been developed. To be effective in a large environment, RPC will have to become a participant in such management architectures.

4.2.6 Interoperability

Different RPC implementations from, say, SUN and OSF share a common vision of the programming interface. They are, after all, trying to make things look as much as possible like a function call. They do not, however, share a common underlying RPC protocol. It is thus impossible for the two to interoperate with each other. [20] RPC cannot, therefore, realistically be considered a truly open standard. The paradigm is certainly standardized, but the actual implementations remain proprietary.

5 Conclusions

What seems most obvious about RPC is that it is a model born in small homogeneous environments. Although it is conceptually satisfying and paradigmatically powerful, RPC does not "scale up" to large enterprise systems very well. Its underlying semantics, while largely invisible to the application, nonetheless cripple actual operations in multi-thousand node networks. The inability of existing RPC products to easily span heterogeneous networks is an almost insurmountable limitation given the extant installed base of legacy networks. Moreover, the session load required by reliable RPCs in the centralized enterprise are cost-prohibitive at best, and technically intractable

[20]It is not a little amusing to note that this situation exists between two organizations that pride themselves as being leaders in "open" computing!

at worst. It is therefore unlikely that RPC will gain ultimate exclusive acceptance in the large enterprise. Until and unless the problems with RPC outlined here are addressed by the vendors, RPC will most probably be relegated to solving more minor, localized problems.

References

[Cor90] John R. Corbin. *The Art Of Distributed Applications.* Springer-Verlag, New York, 1990.

[CS93] Douglas E. Comer and David L. Stevens. *Internetworking With TCP/IP*, volume III. Prentice Hall, Englewood Cliffs, 1993.

[KM91] E.V. Krishnamurthy and V.K. Murthy. *Transaction Processing Systems.* Prentice Hall, New York, 1991.

[RKF93] Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE.* O'Reilly & Associates Inc., Sebastapol, 1993.

[Ste90] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall, Englewood Cliffs, 1990.