To the Graduate Council:

I am submitting herewith a thesis written by Patricia Chapin Price entitled "Network Programming in an Open System Environment". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

William J. McClain, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Patricia C. Price

Date 4/20/94

NETWORK PROGRAMMING IN AN OPEN SYSTEM

ENVIRONMENT

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Patricia Chapin Price

May, 1994

DEDICATION

This thesis is dedicated with deepest gratitude

to my husband and friend

Dr. John K. Price, Jr.

who, because of his constant encouragement and saintly patience throughout the course of my work on this degree, deserves as much credit for its successful completion as I do. His repetition of the phrase "You can do it!" every time I decided to quit was often the only thing that kept me going.

ACKNOWLEDGEMENTS

My primary gratitude goes to the God of the Universe, through whom all things are possible. My heartfelt gratitude goes to my father, the late Richard. T. Chapin, and my mother, Edith S. Chapin, for stressing the value of a good education. I would especially like to thank my husband, John; my mother; my sister, Barbara Chapin; my sister and brother-in-law, Sandra and Jerry Szenfeld; and my father- and mother-in-law, Mr. and Mrs. John K. Price, Sr. for being so emotionally supportive and understanding. My sincere thanks also go to Dr. Sandra Priest who has endured with great patience and understanding an often tired and irritable friend; her encouragement has been invaluable. I would like to thank my employer Martin Marietta Energy Systems, Incorporated for providing many of the resources necessary to complete a Master's degree. I would also like to thank my supervisors over the years, Bill Kiser, Sara Jordan, and Dan McDowell who have all been extremely helpful and understanding. My gratitude goes to SAMS Publishing; Prentice-Hall, Inc.; O'Reilly and Associates, Inc.; McGraw-Hill, Inc.; and the Open Software Foundation for granting permission to use material from their publications in the preparation of this thesis. My extreme thanks go to Dr. Bill McClain, Dr. Tom Dunigan, Dr. Sara Jordan, and Dr. David Straight for: (1) serving on my thesis committee; (2) all the time they have put into reviewing and commenting on the multiple drafts of this document; and (3) the encouragement, support, and technical input they have given all along the way. My utmost gratitude goes to Dr. Bill McClain, who has been not only my professor and the chair of my thesis committee, but has also been a friend as I have trudged through the completion of this degree.

ABSTRACT

This research focused on the concept of "open systems" and the influence that the movement toward open systems is having on network programming and vice versa. Background research focused on the most widely-accepted "open" networking protocols: the Transport Control Protocol/ Internet Protocol (TCP/IP) and the Open Systems Interconnection protocol (OSI). Within those protocol suites, connection-oriented transport protocols in particular were researched and compared to provide a better understanding of the interoperability issues involved. **Achieving application transport independence, rather than transport commonality, was the central focus of the study. The initial premise of the research was: if transport independence could be gained through flexible software, then the push toward adopting one or two common transport protocols to usher in open systems might become less important. At the very least, the use of transport independent Application Programming Interfaces (APIs) could significantly aid in the transition to a more open computing environment.**

The APIs evaluated for possible use in a transport-independent environment were: the Open Network Computing (ONC) Remote Procedure Call (RPC) from Sun Microsystems; the Distributed Computing Environment (DCE) RPC from the Open Software Foundation (OSF); and the Transport Level Interface (TLI) from

Sun. Experimental evaluation of the performance of the ONC RPC and the TLI interfaces was completed for comparison purposes. To accomplish this task, blocks of data of increasing size were requested by client software and sent by the server to the client. The packet data information was captured and evaluated. At smaller message sizes (from 1 to 4 KB) there was very little difference in performance (although RPC transfers required more packets). As the message size increased, however, there was a significant gain in efficiency by the TLI API, making it appear a better choice from a performance standpoint for larger messages. The DCE API was available on only one machine, making network performance analysis of that API impossible.

The ease of development of programs to accomplish the same task (file transfer) was assessed using the ONC RPC, DCE RPC, and TLI interfaces. The ONC RPC interface subjectively appeared to be the simplest to learn and implement. The TLI interface required much more program code development for establishing and breaking client/server connections. The extensive use of pointers in the data transfer made the DCE software seem more difficult than it actually was, a fact which could easily hinder its use.

Several conclusions were drawn from this research: (1) none of the interfaces studied were found on a large number of

disparate platforms, making limited their use to develop applications which were not dependent on a particular transport; (2) the upgrade of the software on one host may make it incompatible with lower versions of software on other host machines; (3) learning to program using the APIs is not a trivial task; (4) lack of availability of network usage tools leaves programmers essentially "programming in a vacuum" and precludes code optimization; (5) obstacles in getting the software installed and operable appeared routinely throughout the research; (6) the number of variables that have to be synchronized to keep a distributed software environment useable are great; (7) claims of "transport independence" regarding a particular API generally mean the ability to employ more than one transport protocol from a single protocol suite (generally TCP/IP); and (8) OSI protocol support is in the early development stages for both the ONC RPC and DCE RPC APIs.

While the importance of API technology in the development of client/server applications cannot be overlooked, no single API is available which can be used to achieve true transport-level independence over a broad base of disparate computing platforms.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

It is virtually impossible to pick up a technical computing magazine or review software and/or hardware specifications without seeing the words "open" or "interoperable"; these two words have become the buzzwords of the early 1990s. There is no doubt that achieving "open systems", whatever they may be, is paramount on everyone's mind.

This study focused on: what "open systems" and "interoperability" are and how they interrelate (Chapter 2); what "open" protocols are and why they are needed (Chapter 3); the Remote Procedure Call (RPC) Application Programming Interface (API) and the value of using RPCs in programming in an open client/server environment (Chapter 4); and the Streams Transport Layer Programming Interface implemented in UNIX System V, as a potential alternative to using RPCs to achieve transport level protocol independence (Chapter 5).

Chapters 1 through 5 contain reference material taken from numerous sources. References pertain to all information preceding that reference. Chapter 6 and the APPENDICES contain original research information.

Development of programs using the RPC and TLI Application Programming Interfaces (APIs) to accomplish the same tasks (file transfer and specific block size transfer) was undertaken. An assessment was made of the performance and ease of use of each API. The main goal of that part of the study was to outline the similarities and differences in using the APIs, assess use of the APIs in the current computing environment, and draw conclusions as to possible usefulness of these (or similar) APIs in the future.

CHAPTER 2

INTEROPERABILITY AND OPEN SYSTEMS

DEFINITIONS

The many seemingly divergent opinions about what exactly constitutes an "open system" actually have much in common. Some individuals claim open systems and UNIX are synonymous and that UNIX compliance is required for a system to be open. To individuals holding this view, any UNIX-based hardware, software, communications, applications, etc. is open. Essentially, if it is not UNIX it is not "open". At the other end of the spectrum are individuals who believe open systems will never become a reality. They believe that the predominance of operating systems such as MS-DOS and MVS make the eventual prominence of UNIX impossible. This view is actually more biased against UNIX than critical of open systems.

There are valid reasons why many people favor UNIX in the open environment. UNIX has been the operating system technology that has remained fairly inexpensive while optimally exploiting the price and performance improvements in microprocessors. UNIX allows software vendors to deploy

development investments across numerous computer platforms, thus allowing independent software vendors (ISVs) to enter more partnerships than they could afford to in a proprietary environment.

One of the major forces currently driving UNIX is the availability of relatively inexpensive, high-quality UNIX expertise. Because UNIX is the teaching platform in so many computer education programs, there is a relatively large labor pool experienced in UNIX. Using UNIX, therefore, minimizes the amount of proprietary technical training necessary and lowers development costs. UNIX vendors are able to pass on the lower development costs to users in the form of lower prices.

Users recognize the transferred benefits of standardized UNIX technologies. Just as employing UNIX makes it easier for ISVs to develop and market new applications, it also greatly simplifies user access to new applications. Furthermore, integrating new UNIX technologies into installed UNIX systems is less expensive and risk-prone than integrating new technologies into most proprietary systems. Installed and stable UNIX systems can generally be expanded with minimal disruption. Many proprietary vendors are implementing UNIX into their computing architectures. Even IBM's MVS/ESA suppliers are incorporating UNIX interoperability. In the past few years, IBM has provided support for UNIX-oriented

technologies, such as TCP/IP, X-Windows, and the Network File System (NFS). However, in many proprietary environments, UNIX interoperability is expensive and sometimes incomplete. When it is necessary to maintain performance of critical work while experimenting in interoperability, users are generally considering UNIX-based platforms.

The truth about open systems lies on both sides of the argument. The basic concept of open systems is the freedom to choose; therefore, building open systems entirely on UNIX seems unlikely. What seems more likely is that users will find intelligent ways to employ both UNIX and non-UNIX technologies in both currently-installed and newly-developed networked systems. Open systems growth and development is dependent, however, on a new generation of sophisticated users seeking individual solutions to their application requirements; users who want control of their own systems returned to them from the vendors who dominated the computer industry during 1960s and 1970s.

The 1980s saw a move of computing toward the desktop and away from the mainframe environment. This movement is one of the main factors fostering a more open computing environment. All these widely disparate and distributed computers now need to communicate with each other.

"Interoperability" and "open systems" are terms used loosely and often interchangeably. In today's market, one would be hard-pressed to find a vendor who would not say they currently have available (or definitely have plans for) open, interoperable hardware and/or software.

Differences in interoperability and open systems are subtle and how soon interoperability gives way to open systems remains to be seen. New technologies offer the potential to increase productivity, but long-term plans are required not only to increase the utilization of existing resources but also to facilitate the adoption of newer and, hopefully, more productive technologies. While the new must interoperate with the old, this does not mean that interoperability and open systems are incompatible goals. Interoperability is usually a stepping stone to open systems. In all probability, when open systems are realized, they will be built largely on a foundation of interoperability. [28]

The computer industry is going through an unsettling, often extremely stressful, but necessary transitional period. A certain amount of risk-taking is required to achieve interoperability. It is extremely difficult to find "shrink-wrapped" integration from a vendor. If users want more than intermediate levels of interoperability, they either have to take control of the integration process themselves or pay a

professional integrator. However, as these involuntary user-developers overcome the risks and uncertainties involved in integrating various systems, they become more computer-literate and willing to take more chances in exploring application possibilities. In response to the increased computer literacy of users, increasingly complex tools will be required. Customers are not only becoming more informed about technology, they are also becoming more knowledgeable than their suppliers about how technology is, and should be, employed in their company. Computer-literate users have become an extremely strong force in the computer industry and are becoming better able to maintain the availability of their own systems, thus lowering their reliance on individual suppliers.

When introducing open systems, the degree to which the system affects the existing computing environment is extremely important. Introducing new technology into an existing environment is particularly difficult if large portions of old technologies must be replaced. The customer's ability to make changes is usually constrained by existing installations. This is where the concept of using interoperability as a way to implement open systems comes into play.

So what then exactly is interoperability? It is easiest to explain interoperability in general by starting with a common example, the electrical plug. Anyone who has traveled abroad

knows they cannot simply plug a blow dryer or shaver directly into the outlet in many countries. Electrical voltage, outlets, and plugs are different, or not interoperable. Interoperability, therefore, can be defined as dissimilar products working together in harmony. Just as there is no one standard for electrical plugs, there is no one standard for computer connections and communications. One of the predominant problems in achieving computer interoperability is that of enabling multiple computers to talk to each other across multiple protocols. Protocols usually depend on a system's native format (i.e. UNIX computers probably use TCP/IP and Ethernet to connect them). IBM machines generally use SNA and/or Token-Ring; DEC Vaxes use DECnet and Ethernet. PC LANs use Novell Netware, Banyan VINES, IBM's LAN Server, or Microsoft's LAN Manager with token-ring, Ethernet, or ARCnet hardware. The mere existence of several different protocols on a network is generally not a real problem until there is a need, for example, for a UNIX workstation to talk to a VAX or IBM mainframe. When it becomes necessary to internetwork multiple LANs, the shear panoply of protocols can become mind-boggling. For example, computers "speaking" TCP/IP do not understand SNA. Most enterprise-wide networks use more than one protocol. Simultaneously supporting more than one protocol becomes more important and complicated as networks grow in size, number, and location. By the end of 1994, 14 % of all networked systems in the U.S. will run multiple protocols.

Establishing the communication of multiple protocols in order for end users to exchange meaningful information is the real challenge of interoperability. [61]

ACHIEVING INTEROPERABILITY

Since many companies maintain multiple physically- and/or logically-separated networks, the existence of several different protocols may not yet be a problem for them. For example, a company may have an SNA network for IBM mainframes, a DECnet network for VAXes, a TCP/IP network for Hewlett-Packard or other UNIX computers, PC LAN for PC users, a MAC network for Macintosh users, and a voice network for telephone and FAX. Each network, however, requires its own equipment, spare parts, maintenance staff, technical support staff, and operational and maintenance experts. Generally, a person who is an expert on one network is only minimally knowledgeable on another. Companies prefer to combine multiple networks and to benefit from cost savings by reducing redundant spare-parts inventories, the number of maintenance personnel, and the individual expertise levels required.

Assuming interoperability is a way to eventually achieve openness, what, then, are some ways to achieve interoperability? There are two predominant and generally-accepted strategies for achieving interoperability: (1)

maintaining existing networks in parallel and placing gateways at strategic locations to allow users of dissimilar systems to communicate, and (2) establishing protocol commonality by choosing one or two protocols as backbone protocols and accommodating all other protocols to be compatible with the backbone. Due to the number of protocols available and widely-used, the first method of achieving interoperability is most favored for distributed, client/server networks. Backbone protocols must be chosen carefully, since each protocol is generally optimized to a particular environment and will not necessarily scale to another. SNA, for example, is designed principally for terminal and mainframe networks, while TCP/IP is optimized for use in wide area networks (but not local area networks). NetBIOS and NetWare's IPX/SPX are optimized for local area networks. The Open Systems Interconnection (OSI) transport level protocol TP4 is one option for local and wide area networks, but it is not nearly as efficient as TCP/IP's transport level protocol TCP. [61]

A third and newer strategy for obtaining interoperability is to move the interoperability issues from the network to the desktop and, in essence, to generate applications that can run over various available protocols, leaving the option of selection of protocol to the user. One method of achieving this independence, Application Programming Interfaces (APIs), will be investigated in greater depth.

Methods of Achieving Interoperability

Interoperability Through Gateways

A gateway is a combination of hardware and software, which is responsible for converting information carried over one protocol into a form that is acceptable for transport using another protocol. One of the major advantages of using gateways is the reduction in network-related problems, which are typically localized and easier to locate and correct. However, gateways are slow, and users will often see a performance degradation from their use. Also, since native protocols are optimized for their environments, introducing foreign protocols may corrupt or slow the system for its users. Accessing remote systems through gateways requires users to master a different command set for each gateway.

One of the most pressing disadvantages of using gateways is that dissimilar networks must be operated in parallel and personnel must be retained to provide expertise in each of the different protocols in use. For example, a company might maintain an SNA network for terminal and mainframe users, a TCP/IP network for the minicomputer and high-end workstations using the UNIX operating system, and a VINES environment for PC users. To connect UNIX users to the mainframe would require a TCP/IP-to-SNA gateway; to connect the VINES users to

the UNIX users and to the mainframe, a VINES-to-TCP/IP link and a VINES-to-SNA gateway would be required. IBM shops traditionally favor the gateway approach.

Gateways are a single point of failure on the network. All traffic from one system must pass through the gateway on the way to the other system, producing a bottleneck which is a threat to a network's reliability and performance. By operating at the upper layers of the OSI stack, gateways are subject to more overhead than bridges and routers.

Most companies are currently employing the simpler gateway approach to build interoperability, even though this approach is less flexible than that of creating interoperability through a common platform. [61]

Interoperability Through a Common Platform

Establishing a common platform into which all computers can "plug" is a newer and more difficult strategy than is that of using gateways. In a platform or transport approach, one or two protocols are chosen to be the protocols used on the enterprise-wide backbone; other protocols not "suitable" for enterprise-wide networks are encapsulated or translated before being transmitted to the backbone.

In theory, the transport approach reduces the number of protocols and parallel networks in use, which simplifies the network in general. Simplified networking eases management responsibilities by not only reducing the potential for problems but also reducing the cost of maintenance. However, no standard common protocol has been universally accepted. This is the "bleeding edge" of technology and is much more fraught with danger than maintaining the status quo.

In the long run, merely because fewer technologies need to be supported, using a single network with a reduced number of protocols would lay a common groundwork on which to add applications and services, simplify network management, and reduce costs. However, migrating existing networks to an enterprise-wide network with a reduced number of protocols is extremely difficult, and disruptions to users may not warrant the benefits gained. Therefore, the most logical approach is to strike a balance between the number of protocols on the backbone and the network's flexibility and speed. While having fewer protocols on the backbone makes management easier, performance may suffer. Encapsulating one protocol into another is much slower than using multiprotocol routers to route several protocols. Protocols suitable for an enterprise backbone might be TCP/IP, DECnet, SNA, and OSI. A protocol such as NetWare's IPX/SPX is not efficient on a WAN because of its routing update procedure. NetBIOS does not include routing

functionality at all and must be encapsulated or translated into a WAN protocol. Appletalk, although it has a facility for routing, requires a great deal of overhead due to its frequent updates to router address tables, and this overhead can very quickly clog a narrow-bandwidth WAN pipe.

Protocols such as TCP/IP, DECnet, SNA, and OSI are more suitable for a wide area network, because their routing schemes are better in tune with that environment. It is essential that the selected backbone protocols accommodate not only the bursty nature of PC LAN traffic but also the time-sensitive nature of terminal traffic. TCP/IP, DECnet, and SNA were designed for terminals, but can more easily be adapted to use for LAN traffic. DECnet and SNA are proprietary protocols and their development and deployment remain under the auspices of Digital Equipment Corporation and IBM, respectively.

Often neither a gateway nor a platform approach makes sense as an intermediate strategy toward achieving enterprise-wide interoperability. Sometimes using both gateways and a common platform is more logical as an interim or long-term solution. [61]

Multiprotocol Routing - Applicable to Both Strategies

Routers and bridges are vitally important devices in the construction of heterogeneous and homogeneous networks. Without multiprotocol routers and bridging a network could not easily support multiple, dissimilar protocols. An OSI Network Layer protocol with a routing protocol can be routed; a non-Network Layer protocol cannot be routed. Having a Network Layer sub-protocol enables the protocol's routers to talk to each other, permits calculation of the most efficient path between two points, and enables the communication of status and traffic information. Protocols such as TCP/IP, IPX/SPX, and OSI can be routed because they all have a routing protocol. NetBIOS and LAT, because they are not implemented at the Network Layer, cannot be routed.

Protocols that lack a routing layer must be bridged transparently. Even though SNA does not implement an industry-standard routing protocol, it can be routed. A router determines the best path from a sending station to a destination station, then forwards packets appropriately. Unfortunately "best" can be defined in various ways and could be the path with either the fewest hops, the least congestion, or the fastest link. Different routing protocols implement the "best" path differently.

Currently, although multiprotocol routers are critical when building interoperable networks, the routers themselves are not interoperable. A Cisco router will not work with a Wellfleet router; a Wellfleet router will not work with a Proteon router. Although the router manufacturers may use the same routing protocol, each has implemented its products in a different and incompatible manner, thus locking end users into a single-vendor proprietary solution. The only area of compatibility among router vendors is over T-1 links where router vendors implement the Point-to-Point (PPP) protocol. [61]

Interoperability Using Application Programming Interfaces

Software developers have to choose the network protocols for which they will write their applications. Writing in several network protocols is a time-consuming and resource-consuming process, since the developer must gain expertise in many different network protocols and maintain code for each. Therefore, most developers do not write for multiple protocols; instead they choose only one or two. If they choose a network protocol that falls into public disfavor, their application will almost certainly fall by the wayside as well.

To encourage developers to write networked applications, it is imperative that the development process be made easier. Using

network-programming-specific **Application Programming Interfaces (APIs)** is one way to accomplish this task. With APIs, choosing a network protocol no longer carries the risk of choosing the wrong protocol or making a long-term commitment to resources. APIs liberate the developer from the details of the network protocol actually used for the transport.

APIs hide the details of the network protocols from the programmer, thus reducing the amount of expertise he/she needs and freeing him/her to concentrate on the application rather than the details of networking. This way, there are more applications running on a variety of network protocols from which the end user may choose.

Two popular APIs, the Remote Procedure Call (RPC) and the Transport Layer Interface (TLI) will be discussed further in Chapters 4 and 5, respectively.

Importance of Standards in Interoperability

There are two types of standards: **de jure** and **de facto**. De jure standards are those that are endorsed by official standards organizations. De facto standards are widely-used and implemented but are controlled by a single vendor or group. Examples of de facto standards are: Novell's NetWare,

Internet's TCP/IP, and IBM's SNA. Examples of de jure standards are FDDI (an ANSI standard), Ethernet and Token ring (IEEE standards), and frame relay (an emerging CCITT standard).

Standards are essential to the building of interoperable networks. In the past, computer vendors have locked end-users into their solutions by selling equipment that would work only with other equipment from the same manufacturer. Such practices, which are unacceptable in other areas of business, have been common in computing for years. By using standards-based products, a company wishing to establish or upgrade a network is not locked into a particular vendor's network strategy, development schedule, or business priorities. The network can be built according to the developing company's business strategy, schedule, and priorities. If a particular manufacturer does not make a required product, the product may be purchased from another company which develops products according to the standards being adhered to by the purchasing company.

The use of standard network components allows network designers to build a modular network by mixing and matching components. Development of products according to de jure standards theoretically invites competition, which drives down the prices of products and ultimately benefits the network

buyer. With de facto standards, there may be only a single source for the standard, but there may be a variety of third-party manufacturers. In the case of official (de jure) standards, the specifications are in the public domain, and competition to manufacture products becomes intense. While there are essentially four predominant enterprise networking protocols -TCP/IP, OSI, DECnet, and SNA - only TCP/IP and OSI are considered "open" and will be considered in considerable detail in the main body of this document due to widespread industry acceptance as, respectively, the de facto and de jure open systems protocol suites. [61]

THE MAJOR PROTOCOL PLAYERS

On the way to interoperability and openness one needs to realize that there are currently four major players in the network protocol arena. Before dealing specifically with the OSI and TCP/IP protocols, it is important to explore briefly the viability of these primary enterprise networking protocols (TCP/IP, OSI, DECnet, and SNA) and, due to their extreme prominence, to review how each could possibly fit into an open system environment. [61]

Transmission Control Protocol/Internet Protocol (TCP/IP)

TCP/IP In General

TCP/IP, while not an OSI protocol, is a four-layer protocol suite which roughly follows the four lower layers of the OSI protocol stack (see Figure 2.1). TCP/IP is named after its two most prominent protocols: the Transmission Control Protocol (TCP) and the Internet Protocol (IP). The term TCP/IP actually refers to the TCP/IP protocol suite, which includes applications such as file transfer, remote host login, and mail. TCP/IP is a packet-oriented, layered protocol suite; the different layers communicate only with the layers directly above or below them, thus promoting a certain amount of independence and freedom for the different layers and applications.

TCP/IP was developed in the early 1970's for the Department of Defense (DOD) so that computer users could communicate with other computers located in government and military facilities, at contractors' sites, and in universities. The Internet is a collection of thousands of computer networks that use the TCP/IP protocol suite to function as a single cooperative network.

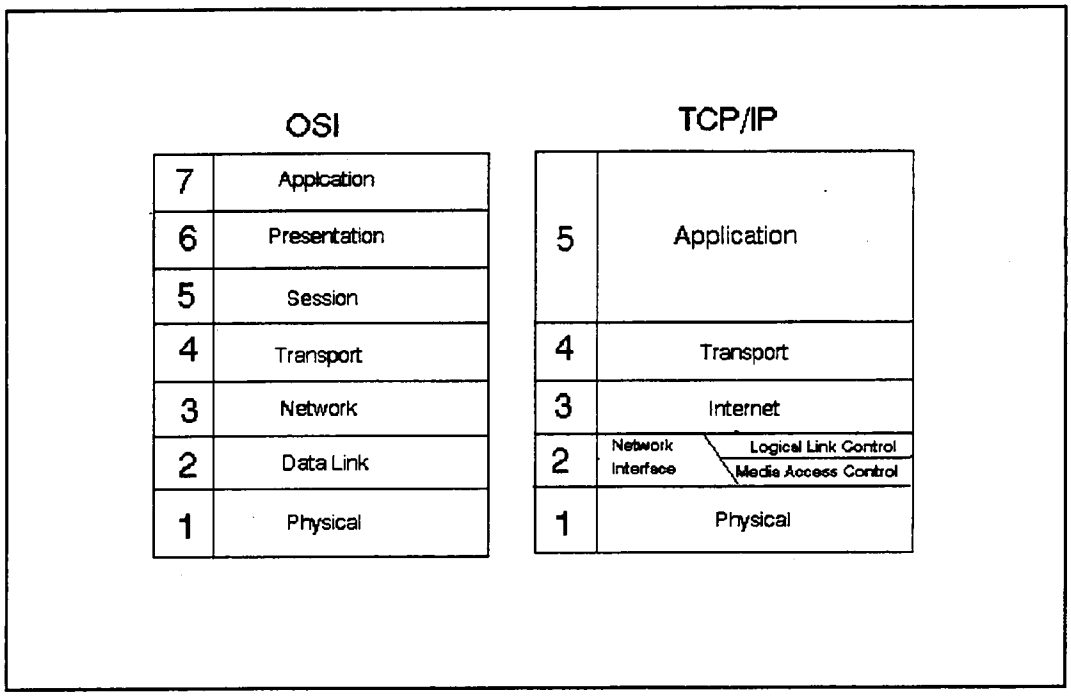

Figure 2.1. The OSI and TCP/IP Stacks.

Because TCP/IP was required for a site or facility to connect to the Internet, many universities adopted TCP/IP for their campus-wide networks. TCP/IP is popular because it is mature, available, and effectively provides interoperability.

Even though it is not an IEEE or ANSI standard, the TCP/IP protocol suite is stable, since one body, the Internet Activities Board (the coordinating committee for the design, engineering, and management of the Internet), controls TCP/IP's development and engineering. TCP/IP is widely available, and hundreds of TCP/IP vendors have implementations for nearly every computer type, from personal computers to Cray supercomputers.

TCP/IP is vendor-independent and, in many cases, free. It allows users to transfer files using the File Transfer Protocol (FTP) software, to exchange E-mail using Simple Mail Transfer Protocol (SMTP)-based mail software, and to log on remotely and run applications using TELNET software. It also provides a facility for managing networks (SNMP). Even though OSI (the official standard for achieving interoperability on a heterogeneous network) offers many of these services, it is not yet completely defined, while TCP/IP currently exists and is being used to solve present-day problems.

TCP/IP's greatest problem is that it has been seen as largely academic and relatively incomprehensible. While TCP/IP, like UNIX, can be difficult to use, it is becoming very popular among corporate MIS departments for internetworking. Although TCP/IP is the most widely-used protocol for heterogeneous networks, it is based on 1970's technology and was designed for minicomputer, rather than personal computer, connectivity. It does not provide optimal connectivity in a network of personal computers. However, third party vendors are quickly bringing robust TCP/IP products to the PC market (i.e. FTP Software's PC/TCP software for the IBM PC).

TCP and IP Protocols

IP refers to the TCP/IP **Internet Protocol**; it is roughly equivalent to the OSI Network Layer protocol. It provides packet routing over different subnets of a TCP/IP network. IP enables applications to operate independently of the media because it hides physical addressing details by using its own 32-bit addressing scheme. IP is a datagram protocol that is capable of discovering the source and destination addresses for nodes on a different network. IP breaks a large TCP packet into smaller pieces, transmits them out over the network, and reassembles the TCP packet on the receiving end.

TCP refers to the TCP/IP **Transport Layer Protocol**; it is similar to the OSI Transport Layer connection-oriented protocols since it provides reliable connections between end nodes. When an application on one network node wants to communicate with an application on another node, the nodes may or may not be on the same network segment. TCP uses a three-level addressing scheme. The first level is the application or program (a process) plus a unique address. A port address is formed at the second level by combining the process address with the address of the originating node. Finally, the port address is combined with a network address to form a unique entity called a socket.

The TCP protocol prepares data for the IP protocol to send in packets. A TCP packet contains the application data plus a TCP header, which consists of addressing and other information. After the TCP protocol has formed a packet, it passes it to IP. IP repackages it into IP packet format. The IP packet, which envelops the TCP packet, is placed inside a logical link control (LLC) packet, which has an address and control header. The whole packet is then placed inside a MAC transmission frame, for transmission out of the node onto the network. When the packet reaches its destination address, it is disassembled as each layer strips off and responds to the information meant for it.

A large amount of overhead is required to provide reliable end-to-end transport. TCP not only must manage the sockets it creates but also ensure delivery of data. When the overhead is too great, the unreliable, connectionless protocol User Datagram Protocol (UDP) is often used. Because UDP is packet-oriented rather than socket or connection-oriented, it demands far less overhead for small messages and is particularly good for simple query-response transactions. For example, the Simple Network Management Protocol (SNMP) uses UDP.

Sun Microsystems (Mountain View, CA) has the most widely-used TCP/IP applications. Sun's architecture, called Open Network Computing (ONC), solves many of the problems imposed by using

TCP/IP. In a comparison with the layers of the OSI stack, ONC includes support for Ethernet, token ring, and FDDI on the Physical Layer. At the Transport Layer, IP is used. At the Session Layer, Sun uses remote procedure call (RPC) tools, and at the Presentation Layer, the Extended Data Representation (XDR) is employed. Application-layer services include the Network Filing System (NFS) for remote file access, remote program execution, and the Yellow Pages for directory services, in addition to the more familiar TCP/IP applications, such as TELNET.

Open Systems Interconnection (OSI)

OSI in General

To speed the migration toward open systems, the United States government has mandated that government networks built after 1990 either provide a migration path to OSI or directly support OSI. OSI is more popular in Europe than in the United States, probably because of a smaller European installed equipment base and the fact that Europe has more countries needing to communicate with one another.

OSI is an international, vendor-independent standard which provides a full suite of more sophisticated services than are found in TCP/IP. If fully implemented, it would promote

intercommunication among companies, partners, and suppliers. Such interconnections are, however, hampered by addressing, naming, and implementation issues.

At its lower layers, OSI supports Ethernet, token ring, FDDI and others as well as LLC, X.25, and ISDN. Network, Transport, and Session Layer services are provided but are not widely implemented. Application Layer services include file transfer, terminal emulation, directory services (X.500), and mail (X.400). To date, OSI has not been widely implemented, although application services, such as directory services and mail, show great potential. OSI protocols are generally CPU-intensive, making the suite less suitable for PC LANs, but more acceptable for host systems. The OSI suite includes specifications for the seven-layer OSI model. The lower-layer standards are fairly widely implemented and often correspond to ANSI and IEEE standards. OSI implementations on the Network, Transport, and Session layers, are less widely-used.

Unlike TCP/IP, OSI tends to be driven by vendors. For example, AT&T has based its Stargroup system entirely on OSI protocols, making AT&T users the largest installed base of OSI in the world. On the other hand, end users often drive the migration to TCP/IP, since it provides a readily deliverable service. [61]

Explanation of GOSIP

End users and vendors are not adopting OSI for a reason. For some, the final impetus to move to OSI could be the desire for standards-based, interoperable networks and products and, more specifically, the vendor-independence and lower costs that result.

The Government Open Systems Interconnect Profile (GOSIP) version 1 mandates that "GOSIP shall be used by Federal Government agencies when acquiring computer network products and services and communications systems or services that provide equivalent functionality to the protocols defined in the GOSIP documents." [61] The key words are **equivalent functionality**. The FIPS document continues: "For the indefinite future, agencies will be permitted to buy network products in addition to those specified in GOSIP. Such products may include other nonproprietary protocols, proprietary protocols, and features and options of OSI protocols that are not included in GOSIP." GOSIP is required to become every government agency's "second protocol" in the 1990's, and many agencies are striving to make GOSIP their first network protocol.

GOSIP version 1, which has been in effect since August 1990, does not require complete OSI computing or compatibility.

Versions of GOSIP beyond Version 2 will probably implement the full OSI protocol suite. A larger number of protocols exist at the lower layers of the GOSIP specification. As the specification evolves, more protocols will be added to the upper layers. Currently the main higher-layer protocols are FTAM (for file transfer) and X.400 (for E-mail).

GOSIP is only one specification of OSI. Other countries are specifying other, often incompatible versions. In the United States, most of the interest in OSI has been generated in the government sector, and many large government agencies have developed plans to migrate to OSI-based networks. Many government suppliers, defense contractors, and systems integrators have started using OSI.

GOSIP-compliant products began emerging in 1991, but applications such as Electronic Data Interchange (EDI) will not be possible until 1994 to 1996, or whenever enough corporations adopt OSI to build a sufficiently common infrastructure to make such applications possible.

One of the main concerns of potential users of OSI products is that the products will not be completely compatible with each other. In the United States, OSInet is an organization of Americans who are trying to solve the problem of testing for OSI interoperability. Similar organizations are addressing the

problem in other countries. [61]

OSI in Perspective

For most U.S. companies, migration to OSI will be at the government's insistence. OSI is likely to be explored in the context of an existing system. Nearly every major computer and network manufacturer is supporting OSI in some manner. Sometimes, the existing networking transport can be untouched, and the OSI applications can be loaded on top. This gateway approach may be slow, thus reducing OSI's interoperability benefits. However, running OSI applications on top of TCP/IP, DECnet, or SNA infrastructure may prove to be the most practical implementation and may provide a fairly painless way to become acquainted with the next-generation protocols.

It is unlikely that the market will be able to deliver a "killer application" that will drive users to migrate to OSI. Most likely, a few OSI applications will be installed in the context of an existing protocol. [61]

Digital Equipment Corporation (DECnet)

Digital Equipment Corporation (DEC) has a plan, called Digital Network Architecture (DNA), for specifying how its computers communicate at each layer ; DECnet is the physical

implementation of the DNA. DEC has delivered five phases of DECnet, but Phases IV and V are the most relevant to enterprise networks. DECnet Phase V allows end users to choose between using DECnet, TCP/IP, and OSI services.

Although other types of computers are supported, DECnet is primarily for hosts supporting the DEC VMS operating system, and, while providing alternative support for TCP/IP and OSI, DECnet itself is controlled by DEC.

When DEC introduced DNA Phase IV in 1982, it defined a 16-bit network address that enabled users to build networks of up to 64,000 devices. DECnet IV also included support for Ethernet and offered a routing capability that enabled designers to build hierarchical networks. Phase IV is not based on the OSI model, but instead on DEC's similar, but slightly different, layered approach (see Figure 2.2). Whereas Phase IV supported DECnet protocols, it had only marginal support for the OSI protocols.

DEC announced Phase V in 1987 and began shipping Phase V-compatible products in 1991. DEC also supports token-ring and FDDI in addition to Ethernet and X.25. DECnet primarily runs on computers running VMS or ULTRIX operating systems, although DECnet implementations exist for Windows, DOS, and Macintosh. DEC supports Macintoshes under Pathworks and PCs under LAN

Manager for UNIX. Pathworks also supports Novell Netware and Banyan VINES. Phase V protocols operate modularly, conforming to the OSI model. [61]

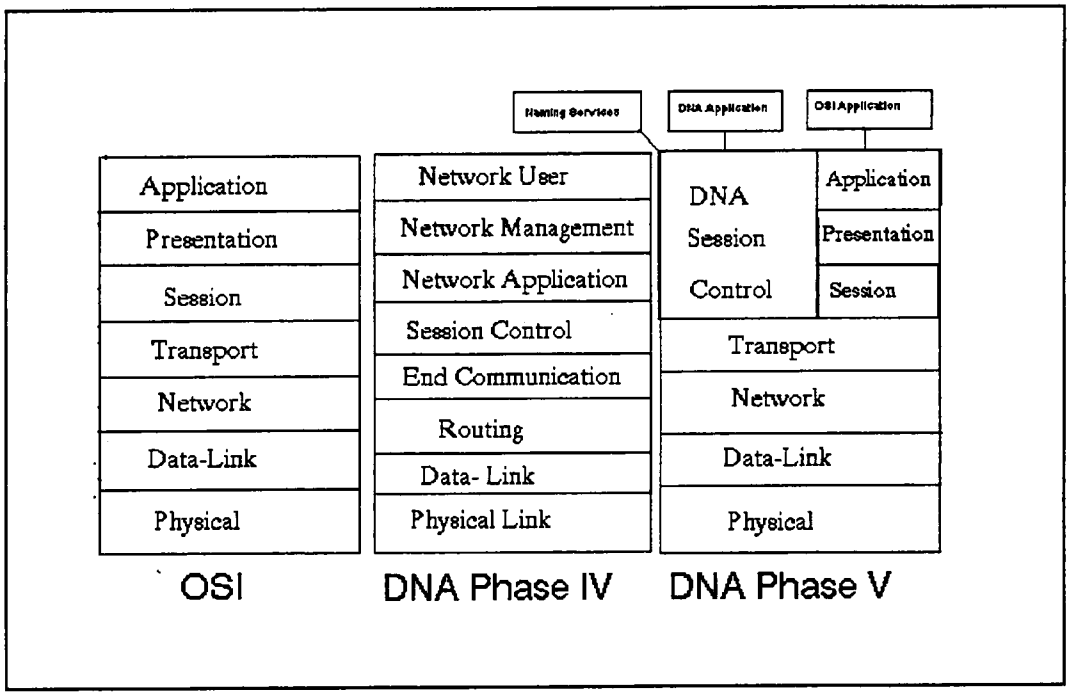

Figure 2.2. **DNA Phase V and Phase V Functional Layers**.

Source: Schnaidt, Patricia. Enterprise-Wide Networking. Carmel, Indiana: SAMS Publishing, 1992. Reprinted with permission of SAMS publishing.

IBM (SNA)

System Application Architecture (SAA)

IBM's System Application Architecture (SAA), announced in 1987, is the company's strategy for uniting its different system architectures (see Figure 2.3). SAA is the set of

standards, development approaches, and software interfaces that govern IBM system design and development. SAA is also the plan for standardizing the interface between the users and the applications, between the programmers and the computers, and among the programs that communicate with each other.

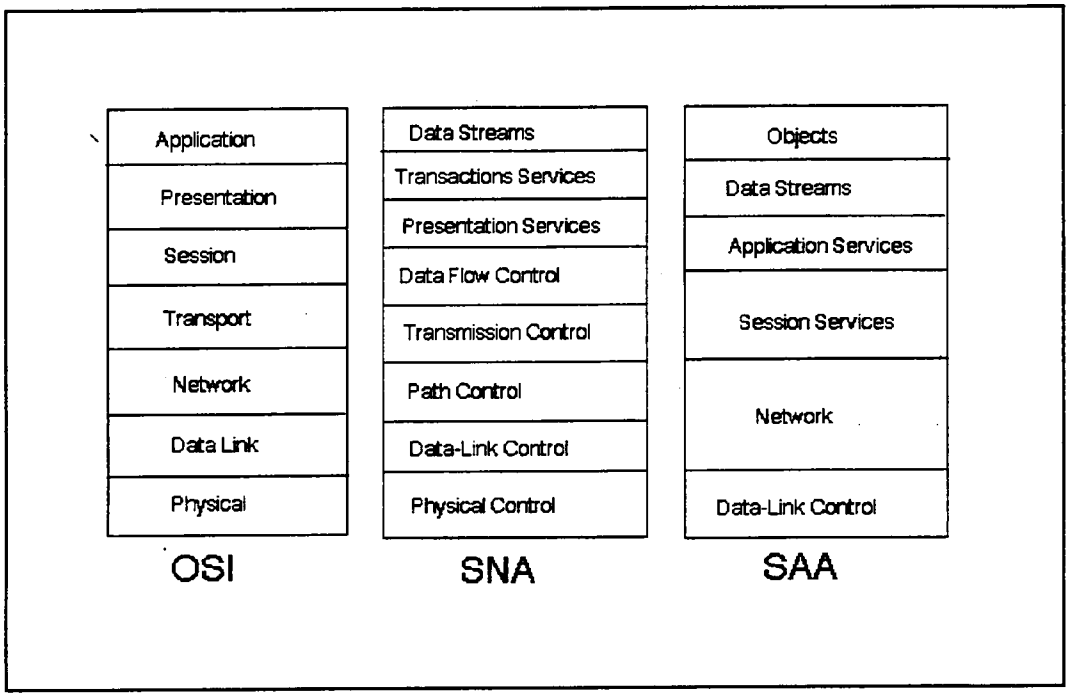

Figure 2.3. **OSI, SNA, and SAA Architectures.**

Source: Schnaidt, Patricia. Enterprise-Wide Networking. Carmel, Indiana: SAMS Publishing, 1992. Reprinted with permission from SAMS Publishing.

SAA provides a complete set of services from data link to applications and provides a model for peer-to-peer computing that is new in an IBM environment. SAA (as APPC) is not widely implemented, and the 3270 protocol is still dominant. Even though IBM has recently begun to adopt such industry standards as UNIX, TCP/IP, and Ethernet, SAA itself is still controlled

by IBM. [61]

SAA represents a significant change for IBM. The computing plan IBM introduced in the 1970's, the System Network Architecture (SNA), followed a strict (master-slave) hierarchy of an intelligent mainframe and dumb terminals via front-end processors (cluster controllers). In SNA, the terminals communicate with the cluster controllers which in turn communicate with the mainframes. All conversations are terminal-to-mainframe. SAA, however, is a peer-to-peer architecture which recognizes intelligent devices other than mainframes. In SAA, applications can communicate directly with each other, without requiring the intervention of a network control device.

Another important feature of SAA is its acceptance of industry standards. IBM's product line now includes FDDI, Ethernet, UNIX, SNMP, OSI, and other industry-standard protocols.

SAA was defined when proprietary network systems were accepted by end users. Advanced Peer-to-Peer Networking (APPN), IBM's architecture for peer-to-peer networks, has not been widely implemented; SNA still dominates. To operate effectively in the current multi-protocol enterprise-network, IBM is having to make some fundamental changes. [61]

Multiprotocol Routers in an SNA Network

Many companies are re-structuring their SNA networks to accommodate PC and UNIX LAN traffic. Unfortunately, the architecture, philosophy, and protocols of an SNA and a multiprotocol network are foundationally different. Since an SNA network is constructed hierarchically, using a set of IBM protocols, it is essentially static. A multiprotocol network, such as a TCP/IP internetwork, is peer-to-peer, uses industry-standard protocols, and grows and changes.

IBM devices are now accommodating multiprotocol networks. Conversely, multiprotocol routers are becoming part of an SNA network, routing TCP/IP, SNA, and other LAN traffic across an enterprise backbone. Multiprotocol routers in general offer greater flexibility at a lower cost than communications controllers. The leading router manufacturer, Cisco Systems, has made significant inroads into the SNA community. The number two company, Wellfleet Communications, also supports SNA. In 1992, IBM announced its multiprotocol router, the 6611, which uses RS/6000 RISC machines. While Cisco's router offers superior functionality in an SNA environment, IBM's entrance into this market is significant.

IBM's 37XX communications controllers function as SNA Type 4 Nodes or routers. Type 4 Nodes use static routing tables that

must be manually configured before the SNA network is brought on-line. A static routing method is inflexible and difficult to manage in a constantly changing multiprotocol network. IBM's APPN supports dynamic routing among its Network Node routers. Although it uses dynamic routing, APPN uses a proprietary IBM protocol rather than an industry-standard routing protocol such as RIP or OSPF.

In order to accommodate non-SNA protocols in an SNA backbone, software can be added to Type 4 Communications Controllers. IBM calls this **protocol enveloping**, but the more common industry term is **encapsulating** or **tunneling**. Although this is not real routing, it does provide the required functionality. [61]

OSI and TCP/IP in the IBM Environment

IBM seems committed to open systems, and, even though SAA is the major architecture for IBM-to-IBM networking, the company intends for SNA and OSI to coexist. IBM and third party vendors offer TCP/IP for the MVS, VMS, OS/2, and AIX operating systems. IBM is developing the TCP/IP base beyond the UNIX-RISC architecture to extend to more "traditional" IBM environments, and support for TCP/IP also extends to NFS and X-Window. IBM is supporting SNMP on a range of systems, enabling its network management product, NetView, to manage

TCP/IP as well as SNA. [61]

THE FUTURE OF INTEROPERABILITY

Understanding the future of interoperability involves understanding two issues. The first issue is the rate of innovation in tools that support interoperability; the second is organizational management. Many of the tools required for heterogeneous and highly-functional interoperability are not available today. Three classes of tools are the most important to aid the evolution to interoperability: (1) networks, (2) computer-aided software engineering (CASE) and (3) systems management. The client/server model is gaining tremendous popularity in interoperability environments and has borrowed heavily from widely-installed software technologies. In the client/server environment, the primary concern is to integrate the best of the relatively stable technologies and to generalize the result in the form of standards. [28]

CASE tools enable sharing, coordinating, and documenting of application resources. Using CASE, users and suppliers can increase program complexity, shorten development time, and improve software quality. The issue of software quality is extremely important. As hardware platforms are connected into interoperable systems, shared software resources become critical to system availability.

Enhanced system management programs will hide application complexity. Eventually, only computers will be capable of keeping up with network changes that may last only milliseconds.

The second issue facing future interoperability is organizational management. Even though new systems-management technologies simplify the demands that interoperability places on staff and support resources, the way that the staff and expertise are deployed will require adjustment. Consoles must be consolidated, data elements must be mapped to each other, and application logic must be made consistent. Organizational issues affect more than just the network management group. Operations, systems, application-development, and storage-management staffs will be required to learn new tools and procedures. The good news is that all of the new skills should be transferrable, and the interoperability phenomenon should feed on itself. While the initial investment in interoperable infrastructures is expensive, subsequent costs of interoperable applications will drop significantly.

There is no computer technology that is appropriate for all problems. The complexity of the computer community requires specialization and diversity. Users can shop knowledgeably for the best alternatives and be engaged in the innovation process that promotes the development of new tools. New perspectives

will be required for procurement priorities, resource allocation and sources of value. New patterns of management must be developed and incorporated into future automated management tools.

Open systems will, in all probability, become the pillar of the computer industry. Rather than wait passively for that day, it makes good sense to reach for interoperability. [28]

CHAPTER 3

OPEN SYSTEMS PROTOCOLS

THE NEED FOR OPEN PROTOCOLS

Assuming the ultimate networking goal is to move toward open protocols, what are open protocols and why are they necessary? Physically connecting computers is not enough to achieve actual information sharing. Computers must adhere to a common set of protocols if they are to interact with each other. [56] As was mentioned in Chapter 2, each computer manufacturer began by defining its own protocol suite so that its own computers could communicate. These groups of protocols are termed "proprietary" because they are controlled by a particular vendor. With a proprietary protocol suite, the vendor can change definitions and implementations at will (and can make every installation "standard").

As time went on, reason dictated that a single non-proprietary protocol suite was required to achieve true information sharing. The protocol suite needed to be "open" so that one vendor would not have an unfair competitive advantage in the market. Over time, the two non-proprietary protocol suites that have emerged are: the Internet suite of protocols,

sponsored by the Department of Defense (DOD), and the OSI suite of protocols, sponsored by the International Standards Organization (ISO). Development of both of these protocol suites started in the late 1970's. Since that time, the Internet suite has been widely deployed, mainly because the Internet researchers made open systems a reality through limitation of the problem, adequate assessment of the technology, and implementation of a set of sound engineering decisions. [56]

The OSI suite was at a tremendous disadvantage from the start and could not achieve focus. Since the development process required international comment and approval, experts from every continent contributed to the standards process and, naturally, disagreements abounded. In 1988, ten years after the introduction of the OSI suite, there were still only pilot projects. However, the Internet suite of protocols has many limiting features making the long-term favorite the OSI suite.

Open systems force the communications aspects of computers to be viewed as a commodity. It is this aspect of open systems which is the most appealing to users; they no longer need to worry about one piece of equipment talking to another piece of equipment on the network. Cost-effectiveness is increased through planning, and competition is enhanced. Open systems are particularly attractive to vendors, because (in theory)

communications development can be focused on one set of protocols, thus making better use of engineering, marketing, and sales resources. [56]

DEFINING OPEN SYSTEM PROTOCOLS

Open systems have been defined as those systems which comply to the Open Systems Interconnection (OSI) standards. A standard is a document which carries the force of the sponsoring entity. There are two organizations which are viewed as having International authority for standards: the ISO/IEC and the Consultative Committee for International Telegraphy and Telephony (CCITT). International standards groups, however, have no force of law. Each national government must determine which standards will be required and/or applied for a particular application.

The members of the International Organization for Standardization and International Electrotechnical Committee (ISO/IEC) are national standards bodies, of which the United States' ANSI (American National Standards Institute) and the United Kingdom's British Standards Institute (BSI) are members. The ISO/IEC also maintains a liaison with other multi-national standards organizations such as the Institute of Electrical and Electronic Engineers (IEEE) and European Computer Manufacturer's Association (ECMA). Documents produced

by the ISO/IEC are called International Standards.

CCITT members are from national Post, Telephone, and Telegraph (PTT) administrations of the European community. There is one voting member per country. If one entity in a country does not provide this function, an agency of the national government provides a delegate to the CCITT. Since the United States does not have a PTT, the State Department represents the U.S. in the CCITT. The CCITT is a division of The International Telecommunications Union (ITU), which is under the United Nations. Documents produced by the CCITT are called recommendations, because the CCITT recommends them to member bodies that implement documents. The CCITT uses a cooperative process to develop the recommendations, and, since members of the CCITT have global coverage, recommendations have had much more impact than the standards produced by the ISO/IEC.

The ISO/IEC and CCITT together produce standards and recommendations which are technically aligned on matters of common interest. Both, however, publish their own versions as standards and make quite a bit of money selling their copyrighted standards which are not as freely available as comparable TCP/IP standards. [9]

Functional profiles have been developed because each standard often contains many more options than can be implemented all

together. If each vendor implements only a subset of options, there is no guarantee that the same subset will be implemented by any two vendors. Systems would only be interoperable in theory. An important step is to identify a common subset of related options and practices that can be used effectively. [56]

In the U.S., the National Institute of Standards and Technology (NIST) hosts the OSI Implementors' Workshop which meets quarterly to produce functional profiles of OSI. The Workshop annually produces a set of Stable Implementors' Agreements. Vendors have sole discretion as to whether or not to align products with functional profiles. [56]

The final step necessary to achieve leverage for Open Systems is a national policy that mandates the use of functional profiles. In the U.S., the U.S. Government OSI Profile (GOSIP) [USGOS88] provides the focus for federal users of OSI. The GOSIP document, a U.S. Federal Informational Processing Standard (FIPS), carries the force of law for Federal users. After a grace period of a few years, newly-purchased computer equipment must conform to GOSIP. GOSIP has no direct standing with non-Federal entities like regional and local agencies. Many agencies in early 1989 were moving toward using GOSIP. [56]

COMMUNICATION USING OPEN SYSTEMS PROTOCOLS

Because this paper focuses on connection-oriented transport protocols, connection-orientation will be the primary focus of this discussion. However, both connectionless and connection-oriented operations are basic to any communications protocol. [2] The primary differences in the connection-oriented and connectionless modes are: connection-oriented uses connection establishment, connection release and expedited (urgent) data, where connectionless does not. All other functions discussed are performed by both connection-oriented and connectionless services. [74]

In connection-oriented operations the user and network set up a logical connection between the source and target machine before the transfer of data is possible. Generally, some type of relationship is maintained between the data units being transferred through the user/network connection. In connectionless operations, no logical connection is established between end stations prior to data transmission and data units are transmitted as independent units.

Connection-oriented protocols provide for acknowledgement of all data units and ensure that data arrives in proper order at the final destination. Mechanisms for retransmission of faulty units are provided if problems occur during transmission.

Connectionless protocols are more robust than their connection-oriented counterparts; data units may take separate paths to avoid failed nodes or congestion points in the network.

A network manager generally implements both connection-oriented and connectionless layers in a system. The choice depends on the type of service needed by the end user. However, if there is any concern for the reliability of data transfer, there will be a connection-oriented protocol available on the system. [2]

THE OSI INTERNETWORKING SERVICE

OSI **communication services** are grouped into **internetworking** and **interworking** services. Internetworking deals with the physical connectivity between systems in different sub-networks. Its major functions are routing and relaying. Interworking performs the communication functions required for applications to communicate meaningfully with each other. The lower three layers of the OSI protocol suite (i.e. 1-3) provide internetworking services. Layer 4 provides an end-to-end reliable transport service to users: the upper three layers compose the interworking service. The OSI Reference Model provides the framework within which protocol standards

are specified for each layer. OSI architectural concepts represent independent conceptualizations of the layers and protocols associated with each layer. The OSI architectural concepts are found in ISO/IEC 7498, which only introduces the architectural concepts; it does not specify a protocol.

The OSI standard considers the network to be composed, conceptually, of a set of subnetworks connected by an intermediate system (IS) and populated by end systems (ESs). Applications reside on the ESs; ISs provide connections among ESs. The communication software necessary to provide both interworking and internetworking services from ES to ES is very complex.

This paper focuses on the Transport Layer of open systems. **The Transport Layer** provides reliable end-to-end transport service to users, using either the Connection-Oriented Transport Service (COTS) or the Connectionless Transport Service (CLTS). In connection-oriented mode there is a full-duplex transmission between two transport service users. Transport functions invoked by the Transport Layer depend on the underlying network type. In OSI there are five different types of transport protocols (TP0 through TP4). Although the Transport Layer provides connectionless service, all existing OSI application protocols are connection-oriented. CLTP (Connectionless Transport Protocol), the ISO protocol used to

provide CLTS, is similar to the TCP/IP protocol suite's connectionless protocol UDP. [74]

COMPARISON OF TCP AND TP4

In a comparison of the TCP/IP and OSI stacks, a few points of emphasis are important. Several organizations and many people have stated that some of these layers are essentially functionally equal. Reports and articles have stated that TCP and TP4, as well as the TCP/IP network protocol IP and the OSI network protocol Connectionless Network Protocol (CLNP) do the same things. While TCP is similar to many of the ISO/CCITT Transport Layer operations, TCP and TP4 differ in many of their features, and, in several instances, capabilities in one protocol are not found in the other (see Table 3.1).

Because of its sheer complexity, the effects on the network of supporting TP4 instead of TCP may have major consequences in relation to throughput, delay, congestion, flow control, time-outs, retransmissions, etc. [2]

TABLE 3.1

COMPARISON OF TCP AND TP4

TP4	TCP
Connection-oriented	Connection-oriented
Complex, many functions	Simple, relatively few functions
Complex and varied message format	One format for segment
Data placed in specific data units from upper layers and sent as Transport Protocol Data Units	Data sent on flow basis from upper layer and sent as segments
Expedited data may arrive out of sequence with earlier	Urgent data stays within order within the stream and segment
Uses OSI service access point (SAP)	Uses 32-bit IP address and port number to achieve socket
push function non-existent	Uses push function to force transfer
One of multiple classes	No class concept in TCP/IP
Uses OSI-based service definitions	Uses TCP-specific primitives
Uses TP-specific timers and state diagrams	Uses TCP-specific timers and state diagrams
No graceful close	Supports graceful close
Two-way handshake connection release	Three-way handshake connection release
May negotiate services	No negotiation of services

Adapted from: Black, Uyless. TCP/IP and Related Protocols. New York, New York: McGraw-Hill Book Company, 1992.

DATA COMMUNICATIONS STANDARDS ACCEPTANCE

By 1990, many large enterprises (e.g., commercial banks, insurance companies, government departments) had either written, or were planning to write, a plan to migrate to the OSI standards. Every major communications vendor is planning to support one or both of the following: migrate away from vendor-specific products toward OSI and/or TCP/IP suites and/or maintain vendor-specific layers and provide interfaces between these layers and the OSI and TCP/IP suites.

A gateway has been designed by the Department of Defense (DOD) to provide the transition between the TCP/IP and OSI protocol suites at the internetwork level. There are two ways of achieving TCP/IP to OSI interoperability: (1) dual TCP/IP and OSI protocol hosts and (2) TCP/IP to OSI Application Layer gateways. The movement to OSI will require transition aids which act as converters between different layers in the stacks and will require an extended period during which different protocols exist and may interoperate with each other.

While OSI is a concept which has been accepted, the TCP/IP protocols have a head start in use and acceptance by vendors and users. Many manufacturers are having to build a product to meet a customer's needs by designing it to have "hooks" into and out of both the TCP/IP and OSI worlds. This approach is

not contrary to the spirit of OSI and is not an issue addressed in TCP/IP. In OSI, it does not matter what happens internally in a system as long as a given standardized input into the system produces a predictable and standard output from that system.

As of this writing, TCP/IP and the OSI standards are garnering the most support for use as international data communication network standards. [2]

CONCLUSION

As one may gather from this chapter and the previous one, the entire concept of what constitutes an open system and how (or if) one would and/or could achieve implementation of a network environment adequate to support open systems is a matter of great debate and will certainly provide a fertile area for future research and innovation.

CHAPTER 4

THE REMOTE PROCEDURE CALL (RPC)

RPCS DEFINED

One way to move toward a more open computing environment may be the use of Application Programming Interfaces (APIs). The Remote Procedure Call (RPC) API will be discussed will be further in this chapter. This discussion covers the RPC facility developed by Sun Microsystems' Open Network Computing (ONC) group and the Transport Independent RPC (TIRPC) software developed by Sun. Also covered are elements of the Distributed Computing Environment (DCE) computing standard that defines another flavor of RPC. The Netwise software for performing RPCs will also be discussed.

With Sun's ONC RPC 4.0, a local-to-remote connection is created using either the TCP or UDP IP transports. With Sun's latest transport-independent RPC (TIRPC), transport selections can be made at run-time to accommodate client, server, and user preferences. ONC's RPC forms the foundation of most distributed system utilities used today (i.e. NFS and NIS). [4]

RPC implementations can be generalized to look similar, totally independent of the machine's operating system, permitting a programmer to learn the art of remote procedure calling under UNIX and then to apply those techniques to any other RPC system.

Remote procedure calling is the ability to call procedures outside of an application's current address space, thus allowing a local client program to execute a procedure remotely, pass data to the server process, and receive the result back. RPC calls allow distributed applications to be written which utilize the computing resources on the network. An RPC system is a logical component of a distributed computing environment and is the collection of software necessary to support remote procedure call (RPC) programming and the necessary run-time services.

Remote does not, however, necessarily mean communicating over a network. Although clients and servers are often thought of as separate machines on a network, it is more accurate to think of them as processes running anywhere on the network. Any application may take on the roles of both client and server. In the X-Window and RPC systems, the client and server software may, and often does, exist on the same machine.

Client/server computing is becoming extremely popular in a

distributed processing environment where one entity requests work to be done and another performs the work. Both the X-Window and RPC systems are examples of client/server models for distributed computing. While both provide the ability to execute programs remotely, the RPC system communicates more directly and with less overhead.

The X-Window system and RPC use the terms client and server differently. In X, remote client applications communicate with the local display server. In RPC, the local client application makes procedure calls that can be executed on remote servers. [4]

When using an RPC tool, a program is divided into a client module and a server module; these modules may run either on the same or on different machines. The RPC tool hides the actual location of the server from an application, allowing the client to call any server module just as if it were written in the same code.

To use an RPC tool, the programmer writes an application, either as a whole program or a client/server program. The programmer then writes the RPC protocol specification containing the declarations for all the server functions that will be called by the client and for all the external variables, as well as for information on how the client

locates the server. Using the RPC compiler, the specification is processed twice, once to generate the source code for the client and a second time to generate the source code for the server. The programmer then links the client and the server code to build executable files. [61]

A simple RPC application would consist of a single client communicating with a single server machine to execute a procedure. A **synchronous** remote procedure call is one in which the client application communicates with the requested procedure on the remote server by transmitting arguments to the server and later retrieving the results. In other words, the procedure execution must be complete before control returns to the client.

RPC programming interfaces hide the actual location of the called procedures by using a request and reply communication model. The client procedure transmits a request message to the server procedure which returns a reply message. Client and server processes communicate by means of two **stubs**, one for the client and one for the server. The stub is the communication interface which implements the RPC protocol and specifies how messages must be constructed and exchanged. Stubs are generally generated by a protocol compiler and are eventually linked with the client and server programs.
At the server side, the RPC tool generates server stubs for

each remote function. These stubs are linked with remote procedures and a dispatcher procedure to create a server module. On the client side, the RPC tool generates a client stub that acts as a replacement for the procedure.

When the client calls the remote procedure, the client stub gets the call, packages the request, and sends it to the server module using the network protocol (i.e. TCP/IP, IPX, or Appletalk) which was linked in during compilation. When the dispatcher at the server receives the remote procedure call, it hands the request to the proper server stub. The server stub disassembles the packet, executes the request, and returns the answer to the dispatcher, which returns the answer to the client. The existence of the network becomes transparent to the application.

Local calls are mapped by functions in the stubs into a series of network function calls. The stubs also contain procedures which use the library to locate a remote process. The client calls the procedures in the stub. The remote server process listens to the network through the server stub, which uses a local low-level procedure call interface to perform low-level functions.

THE RPC ARCHITECTURE

The RPC architecture consists of three parts: a **program to resolve addresses**, the **procedure** on the server machine, and an **application** on the client machine. For illustration purposes, the component elements of the ONC RPC architecture will be discussed in more detail.

Program to Resolve Addresses

The key to all RPC operations is the program that resolves addresses. On UNIX System V Release 4 (SVR4), that program is called ***rpcbind*** (previously called ***portmap***). The ***rpcbind*** daemon process runs on the server machine and maintains a table that maps available RPC procedures into their corresponding transport addresses. The transport address of the ***rpcbind*** process on the server machine must be known by the application on the client machine. The application on the client machine must first send a query to the ***rpcbind*** process on the server machine to obtain the ***transport*** address of the procedure. To allow clients to identify a particular server process, both TCP and UDP have a defined group of "well-known" ports. The ***rpcbind*** process must attach to a well-known transport address.

As the ***rpcbind*** process begins, it determines which transport providers are on the server's system. For each transport

provider, it attaches to a well-known transport address which is obtained through the Name-to-Address mapping routines. For example, the well-known address of ***rpcbind*** over TCP/IP is the server machine's IP address combined with port 111.

The Remote Procedure

The **remote procedure** is the next element of the RPC architecture. To make a procedure available over the network, the programmer must write the procedure and employ RPC facilities to encapsulate it into a program.

There are therefore two processes running on the server machine: (1) the ***rpcbind*** process and (2) the program containing the procedure. The program containing the procedure is attached to an arbitrary transport address, but that address is known by the ***rpcbind*** daemon. [48]

The Application

Applications on clients call procedures through the following actions:

1. The application issues a call for a procedure on a server machine; selects a transport provider on the client machine; and sends a message to the ***rpcbind*** process on the server machine. The message requests the transport address of the program containing the desired procedure.
2. The ***rpcbind*** process receives the message and notes on

which transport provider the message arrived. It informs the client machine of the transport address of the program containing the procedure. Since the program containing the procedure may have several transport addresses (one for each transport provider on the server machine), the ***rpcbind*** process returns the transport address corresponding to the transport provider over which the query arrived.

3. The application on the client machine uses the transport address to communicate with the program containing the procedure. [48]

Transport-independence is one of the most important advantages of the RPC architecture. Processes on a server machine make themselves available over every transport provider, and the client application may use any transport provider to communicate with the server machine. Applications on the client machine need to know only the transport address of the ***rpcbind*** process; specific server process addresses are obtained by the client through a query of the ***rpcbind*** daemon. [48]

RPC OPERATION

It has already been stated that services make themselves known to a network of clients through an independent naming service or daemon which gives the clients the address necessary to open a communication channel with a server. The server may then accept (or deny) client requests. The connection is closed when the remote session has ended.

Addressing mechanisms differ between socket-based and transport-independent flavors of RPC. Transport-independent RPC keeps addresses (e.g. server/service names, transports client and server addresses) at a symbolic level. Socket-based RPC systems define the default network services for a system in the **/etc/services** file, and they become available at boot time. If the ONC Network Information System (NIS) database server is running, available services are located through it.

The service, identified by the port to which it is listening, is a daemon waiting for a connection request. The ONC network service, ***portmapper***, maps services to ports, an important part of client/server communication. Both the client and server have direct access to a machine's portmap. [4]

Applications using the Remote Procedure Call facility provide a high-level model for client/server interactions, allowing C programs to be written using a normal procedure call model, which permits the invocation of procedures on another machine. Programmers may write powerful networked applications and turn non-networked applications into networked applications. Because most applications are divided into procedures, the programmer only has to determine which procedure(s) should be run on remote machines and modify the application to use the RPC mechanism. [48]

An RPC system alone is not sufficient to permit building a networked application. Additional tools and facilities are generally provided as part of a distributed computing environment. [4]

The major elements of a distributed computing environment are:

1. **Machine-independent data representation tools** to permit machines using different data representations to exchange data.
2. A **protocol** to specify the low-level client and server responsibilities during a remote procedure call.
3. A **protocol compiler** to generate RPC interface stubs, thus isolating the application from the underlying protocol and other low-level network programming.
4. **Authentication services** to ensure appropriate access rights are satisfied.
5. **Network naming services** to permit applications to locate conventional network objects (i.e users and hosts) and bind to registered services on the network.
6. A **network time service** to provide host synchronization to establish a single network time.
7. A **distributed file system** to eliminate file redundancy and provide fast, secure, fault-tolerant file-sharing. [4]

Two of the more prevalent implementations of distributed computing environments are: Sun's Open Network Computing (ONC) and the OSF's DECORUM DCE, based on HP/Apollo NCS RPC. This paper will cover the significance of these two distributed computing environments, concentrating on features that most affect RPC programming. Currently the two implementations are completely incompatible, supporting different protocols and

Application Programming Interfaces (APIs). [4]

DISTRIBUTED COMPUTING STANDARDS

The Open Software Foundation (OSF) is a not-for-profit research and development organization headquartered in Cambridge, Massachusetts with more than 400 members world-wide.

In response to the Open Software Foundation's request for technology (RFT) in 1988, two camps proposed enhancements of their existing distributed computing frameworks. On one side was Open Network Computing (ONC) with AT&T, Sun, Novell, and Netwise supporting new developments on the top of the existing ONC RPC system. On the other side was an enhanced Network Computing System (NCS) from HP/Apollo with backers from IBM, DEC, Apollo, and Microsoft.

At the onset of the decision process, it seemed reasonable to assume that OSF would select the ONC proposal due to its establishment in the market and the fact that ONC RPC source code was made freely available to the public. When it finally made a decision, however, OSF selected pieces from both submissions for inclusion in the Distributed Computing Environment (DCE) and finally selected NCS over ONC RPC as the primary RPC mechanism. However, much of what ONC and NCS/OSF

proposed in their initial announcements was aimed for future release. Neither the complete NCS/OSF nor the ONC distributed computing environments can be obtained today.

Both NCS and ONC, as they exist today, are integral parts of existing operating systems, and development tools and APIs are shipped with the systems. ONC development tools are available from the Internet archives.

While both contemporary ONC and NCS state the same fundamental goals, they are incompatible, and major technical differences exist between the two. [4]

The Network Computing System (NCS)

The Network Computing System (NCS), a particular implementation of the Apollo/HP Network Computing Architecture (NCA), is the software available for a distributed application developer to use (see Figure 4.1). NCS is a combination of two Apollo/HP products: NCS/NCK and NCS/NIDL.

NCK is the Network Computing Kernel, the run-time library that includes RPC support and Network resource-manager service with access routines called the Location Broker. The Location Broker is the naming or mapping service that keeps track of network resource objects (e.g. servers). Low-level RPC network

communication is accomplished through the use of a proprietary Network Data Representation (NDR) format.

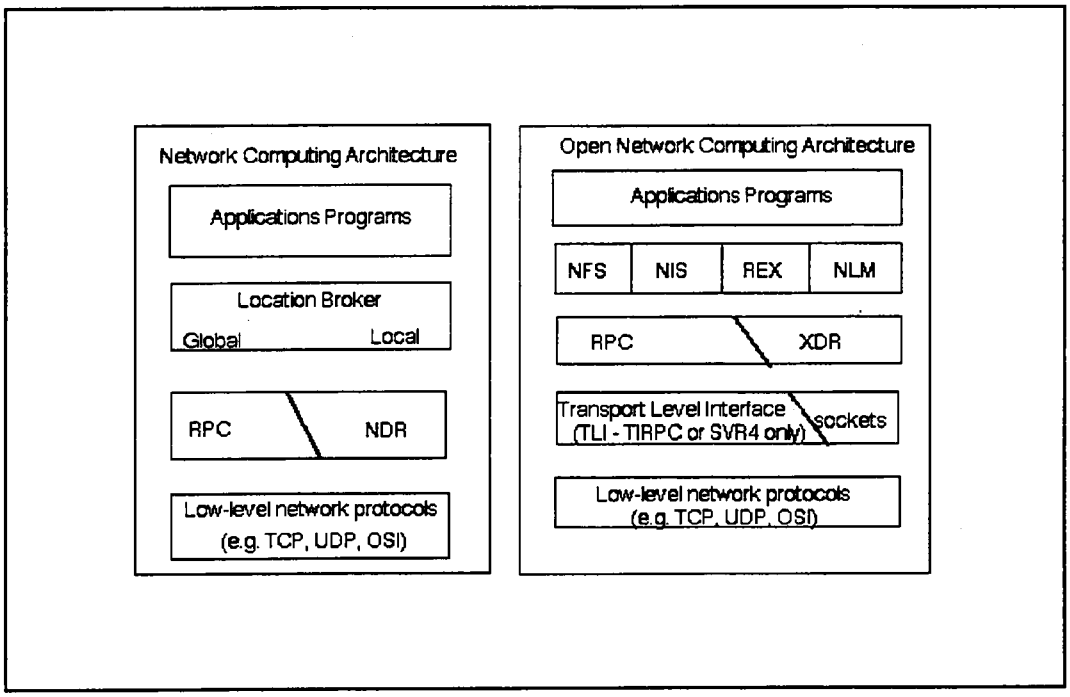

Figure 4.1. **Comparison of ONC and NCA Architectures.**

Adapted from: Bloomer, John. Power Programming with RPC. Sebastopol, California: O'Reilly & Associates, 1992.

The second NCS component, called the Network Interface Description Language (NIDL), is a distributed application development toolkit, including a protocol compiler. NIDL translates interface and protocol descriptions into C code that implement NCS-style RPCs and provide NCK routines for use by cooperating clients and servers.

The NCS/NCK and NCS/NIDL products are supported under Domain/OS, HP-UX, SunOS, VAX/VMS, and VAX/ULTRIX operating

systems. [4]

Sun's Open Network Computing (ONC)

Sun's Open Network Computing (ONC) (see Figure 4.1) refers to a suite of products which includes the publicly-available External Data Representation (XDR) for low-level RPC communication. It also includes the Network Information Service (NIS) (for managing and tracking network resources) and the RPCGEN protocol compiler (for reading an interface description written in RPCL and generating the C code for the client and server). RPCGEN and other ONC RPC support tools are freely available from Sun. ONC also includes a stateless Network File System (NFS) with automatic remote volume mounting, file-locking, and status monitoring mechanisms. Each Network Information Service (NIS) master server manages mapping between network resources and the values for the domain of a machine. Each map corresponds to and is built from the standard system ASCII data files: **etc/passwd, /etc/group, /etc/hosts, /etc/networks, /etc/protocols, /etc/ethers,** and **/etc/netgroup**. Each machine in a domain maintains a slave server, replicating and keeping the maps loosely consistent across the domain.

NFS was one of the first services built on top of ONC RPC. Servers employing the stateless protocol UDP do not have any

sense of previous client interactions. Since the concept of state is required for file-locking, it is accomplished through an additional service, the Network Lock Manager (NLM). [4]

Examining ONC and NCS

Machine-Independent Data Representation

Since data on a remote system may not have the same format as on the local system, differences in format may cause a remote procedure to misinterpret the data. To solve this problem, all data passed to the remote procedure and returned must be encoded in a common data representation. The ONC RPC uses a single-canonical format for data representation known as External Data Representation (XDR). Remotely located clients and servers must communicate using a machine-independent data representation. Client and server stubs translate data into and out of this one form. If a protocol compiler is used, service procedures and the data structures to be exchanged can be specified, and the translation process can be relegated to the stubs. [48]

Current NCS and ONC use different data representation schemes. NCS uses a "receiver-makes-it-right", multi-canonical approach, while ONC uses a single-canonical format.

NCS Data Interchange Format

The Network Data Representation (NDR) allows successful data exchange between systems with different formats. It handles differences like big-endian versus little-endian (byte order), ASCII characters versus EBCDIC characters, and other incompatibilities. A process called **marshalling** is used during a remote procedure call. Marshalling converts data into a byte-stream format and packages it for transmission.

Data transmitted across the network undergoes a process called un-marshalling. If the data format of the sender and receiver is different, the receiver's stub converts the data to the correct format for the system, and passes the data to the application. [63]

Exchange of data between like machines is simplified since no translation is required. A receiving machine requiring a different data representation must invoke the appropriate conversion routine. With NCS, the data may be sent in 16 different formats, with the protocol providing information to the receiving machine as to which filter to use to read the data. If a machine type is added to an existing distributed application, the new requirements for translation placed in the client/server code can cause major library rewrites or recompilation of the application. [4]

ONC Data Interchange Format

Single-canonical forms, while potentially requiring more overhead, do present a more consistent application interface. The ONC XDR uses one intermediary data format and encodes and decodes data irrespective of the machines involved. Encode and decode steps are required even when compatible machines are communicating. Even though this places a burden on the send and receive processing, all applications know the format in which the data exists. When big-endian (MSB first) machines are communicating, the impact of the big-endian byte-ordering used by XDR is small. Significant overhead is introduced, however, when two small-endian machines communicate, since bytes must be re-ordered at both the sending and receiving ends to accommodate the XDR single-canonical order. In the future, ISO/OSI standards should bring the NCS and ONC standards closer together. Both NCS and ONC are moving to accommodate the ISO/OSI ASN.1 Basic Encoding Rules (BER) single-canonical standard, while maintaining backward compatibility. [4]

NCS and ONC are also moving to permit migration to ISO/OSI network libraries by way of compile-time switches in the protocol compilers. Today, RPC libraries are bundled with the operating systems reference low-level TCP/IP and proprietary network protocol libraries. Both NCS and ONC will maintain the

ability to use a standard low-level network communication library while including transport-independent communication in the future. [4]

Transport Protocol Support

ONC RPC 4.0 supports TCP and UDP. The current NCS suite supports UDP transport and the proprietary Apollo/HP Domain DDS Transport. [4]

ONC RPC AND TIRPC

Sun has developed a Transport Independent RPC (TIRPC) interface which will be discussed in greater detail later in this chapter. Under TIRPC it is not necessary to specify a transport protocol when writing protocol definitions of client or server code. The structure of the code is the same, whether connectionless or connection-oriented is selected. TIRPC sits on top of the network transport protocol and provides run-time independence by using the Transport Layer Interface (TLI) in UNIX System V Release 4 (SRV4) (see Chapter 5), thus hiding the RPC mechanism from the underlying transports. The name-to-address translation and network-selection libraries select the protocol and perform necessary associations or look-ups to find the server processes.

True transport independence is one of the most important additions to ONC. The System V Transport Level Interface used in ONC's new release is the same application programming interface as the X/Open Transport Interface (XTI) interface. There are already 800,000+ TLI installations. The new ONC distributed computing environment is also now available for SunOS 4.x as TIRPC. [4]

NCS RPC

The current NCS RPC system specifies the Transport Layer, making it dependent on which protocol is in use (e.g. UDP datagrams or the Apollo domain DDS datagrams). Plans have been announced for accommodating ISO/OSI TP4 and DECnet transports, but such levels of transport independence have not yet been announced. [4]

Future Directions

HP/Apollo plans to add many of the same features ONC has promised. By adding additional, non-proprietary transport protocol support, they hope to address a broader UNIX network market. There are plans to improve the integration of lightweight multiple threads of execution into the standard product. Extensions to speed bulk data transfers that cross packet boundaries are also planned, and a single-canonical

external data representation will be supported. [4]

Concurrency at the Server and Client

The ability to multi-task within the server permits the user to design a concurrent service, which is capable of handling multiple requests simultaneously. On the client side, multi-tasking allows a client process (or thread) to stay blocked while waiting for the normal request/reply cycle of an RPC to complete. Other processes might be executing asynchronously to the I/O operations. Once a reply is received from the server, execution within the process can be terminated or branch somewhere else, thus allowing a client to make multiple requests at the same time.

There are two ways to accommodate this multitasking: **heavyweight** and **lightweight** processing. Heavyweight replicates the complete process context, including the entire address space, forking-off another copy of the current program. Lightweight processing begins another thread of execution in the same process address space. Shared code segments must be re-entrant, thus allowing multiple threads to execute the same code segment without unintended side-effects. Neither NCS RPC nor ONC RPC currently support any form of multi-tasking at the client.

Heavyweight Server Processing. ONC RPC, through the use of a protocol compiler, supports a brand of **fork()**-based servicing which is compliant with the Internet superserver **inetd**. Neither NCS nor ONC support generic heavyweight processing within the server. Each programmer must add his/her own heavyweight processing to service procedures or change the compiler-generated service dispatch procedure. Heavyweight multi-tasking is often slow, inefficient, and resource-intensive.

Lightweight Server Processing. Lightweight processing is recommended for an information server to perform reasonably well under load. The X server, for example, has its own internal lightweight process package.

There are several mature lightweight processing packages: Sun has the LWP package; HP/Apollo has the Concurrent Programming Support (CPS) library; and DEC has CMA, the foundation for OSF DCE's non-OSF/1 threading support. The POSIX draft includes pthreads, and most lightweight processing libraries are moving toward the pthreads solution.

NCS (through the NIDL compiler) can generate re-entrant server code to interface with CPS where available, though not all operating systems can support CPS. ONC RPCSRC and TIRPC libraries are not re-entrant, forcing the programmer to get

below the socket-watching and I/O performed in the ONC library to avoid threads stepping on each other. [4]

The Importance of Protocol Compilers

A protocol compiler is extremely important for writing distributed applications. With it, code is generated to perform the required functions. The protocol compiler is often the only part of a distributed computing environment about which one needs to be concerned.

Both NCS and ONC suites include protocol compilers: the NIDL compiler and the RPCGEN compiler, respectively. Both protocol compilers use extensions of C and employ preprocessor directives and block structures to define the protocol to be used between the client and server programs. Programs, procedures, and versions are identified numerically.

An alternative ONC protocol compiler, RPCTOOL from Netwise, provides a number of features RPCGEN does not. RPCTOOL, like NIDL, is a commercial product; RPCGEN is free.

RPCTOOL is not a standalone product and assumes the ONC RPC system is available. It provides full support for all new ONC RPC features, including transport independence, and the ability to select either the ISO/OSI ASN.1 canonical data

exchange format or the XDR standard. [4]

Authentication Services

The NCS by itself specifies no authentication services. The ONC distributed computing environment has three levels of built-in security: (1) none, (2) UNIX-style user name and password, and (3) a Data Encryption Standard (DES) mode. User security and authentication schemes could be implemented in either RPC mechanism. Both environments are moving to support the MIT Kerberos authentication system. [4]

Network Resource Naming Services

NCS uses the Location Broker to perform maintenance and look-up services for network objects. Many people feel that the Location Broker is more sophisticated and robust than ***rpcbind***, possibly due to the inheritance of state from the NCS RPC. The ONC Network Information Service (NIS) needs separate servers to detect server crashing and to enforce file-locking. Both NCS and ONC plan to support the ISO/OSI X.500 directory-naming protocol. [4]

Network Time Service

Both ONC and NCS support the Network Time Protocol for synchronizing time-critical tasks across a network.

Distributed File System

Sun's Network File System (NFS) will be the foundation for the ONC distributed computing environment. NCS does not explicitly include a file system; Apollo and HP machines ship with distributed file systems. The Andrew File System (AFS) was selected by OSF and will be included in the DCE NCS. OSF selected PC-NFS as the best solution for a PC file system. [4]

The Netwise Alternative

Netwise, Inc., a network software vendor with an established and strong presence in remote procedure calling and job entry for mainframes and a rapidly growing commercial UNIX and PC base, seems to have taken the best features of both Sun/AT&T and HP/Apollo, added information from the evolving ISO/OSI and OSF Standards, and designed a number of versions of RPC-based network programming. The environment contains only some of the pieces that the ONC and NCS environments include, while layering upon existing or future standards.

Netwise has specified or developed nearly everything required for distributed application development (from external data representation to the application protocol compiler). Neither a network time service nor a distributed file system, however, is specified by Netwise. [4]

The Netwise distributed computing environment consists of:

1. Network libraries that are different for each machine type.
2. Server control procedures for a number of different server binding and concurrency selections.
3. The RPCTOOL protocol compiler.
4. RPC extensions that perform such tasks as server registration and look-up using the ONC naming services. [4]

The RPCTOOL Version 3 API has a command line switch on the protocol compiler to specify whether the ISO Application, Presentation, and Session Layers (including ASN.1 BER) or ONC RPC's socket transport should be used. The client and server stubs are generated with either ASN.1 BER or XDR single-canonical formatting.

Run-time selection of OSI-based network libraries is supported. Netwise does not specify nor provide the actual source for the low-level networking, but layers on top of publicly- or commercially-available libraries. One problem with this approach is that on heterogeneous machines, a copy of Netwise RPCTOOL, or at least the support libraries, may be required on each machine to generate the necessary client and

server code. [4]

Protocol Compiler

With the RPCTOOL compiler, users can include their own code or choose from a variety of predefined control procedures. [4]

Authentication Services

RPCTOOL, like NCS, specifies no authentication services. User code must be developed to use the standard UNIX security equivalents. Since the current ONC release already includes a DES encrypted security scheme as well as the standard UNIX user name and password verification, low-level XDR calls can be used under RPCTOOL to facilitate ONC-style authentication. [4]

Network Resource Naming Services

RPCTOOL under DOS depends on Novell's Netware product for naming services. Under UNIX, for either OSI or ONC sockets-based communications, the standard **/etc** files are accessed using the standard ***get*** and ***set*** routines. Services at well-known ports can be accessed through the *inetd* Internet super-server or through NIS. Use of *inetd* and the **/etc** files requires root privileges each time a server at a well-known

port is installed or de-installed. [4]

The Application Environment of the Future

There may be a common application environment in the future. The Object Management Group (OMG), still active today, was formed in 1989 and had more than 100 international members in 1992, including end users and vendors of hardware platforms, object-oriented databases, applications, and tools. Members represent different network, operating system, and hardware cultures. Sun, HP, DEC, HyperDesk, IBM, and Architecture Projects Management Limited are members. OMG is not a standards body; it will endorse or establish specifications that make up an Object Management Architecture (OMA). The OMG is attempting to establish a distributed object programming model based on technology that enables access to objects across a network.

The OMA contains four pieces: the Object Request Broker (ORB), object services, common facilities, and application objects. All pieces interact through the ORB, a general-purpose, distributed object management facility. By adhering to the ORB interfaces, the application developer is isolated from the details of the underlying network, operating, and RPC systems. The OMA will provide a higher-level notion of distributed computing than does OSF/DCE or ONC. ORBs can be connected via

gateways, thus allowing applications on heterogenous environments to interoperate. The final goal is source compatibility of applications across different ORB implementations.

In 1991, as a first step toward bridging together their respective RPC systems, HP and Sun submitted a joint proposal to the Object Management Group's ORB Request For Proposals (RFP). A task force was set up to issue the RFP, evaluate the submissions, and select a specification. A common specification submitted by Sun, HP, DEC, HyperDesk, and NCR was endorsed. It included the ability for an application to generate object requests dynamically using information from an interface repository. Full corporate OMG approval is expected. OSF, UNIX International, and a number of commercial vendors may choose to support the OMG-endorsed ORB specification. The X/Open Common Applications Environment plans to include OMG-approved specifications. [4]

HIGHER LEVEL PROGRAMMING WITH TIRPC

TIRPC Defined

Transport-independent RPC (TIRPC) libraries attempt to isolate the application from the transport used, making it easier to program for use of a variety of transports (see Figure 4.2).

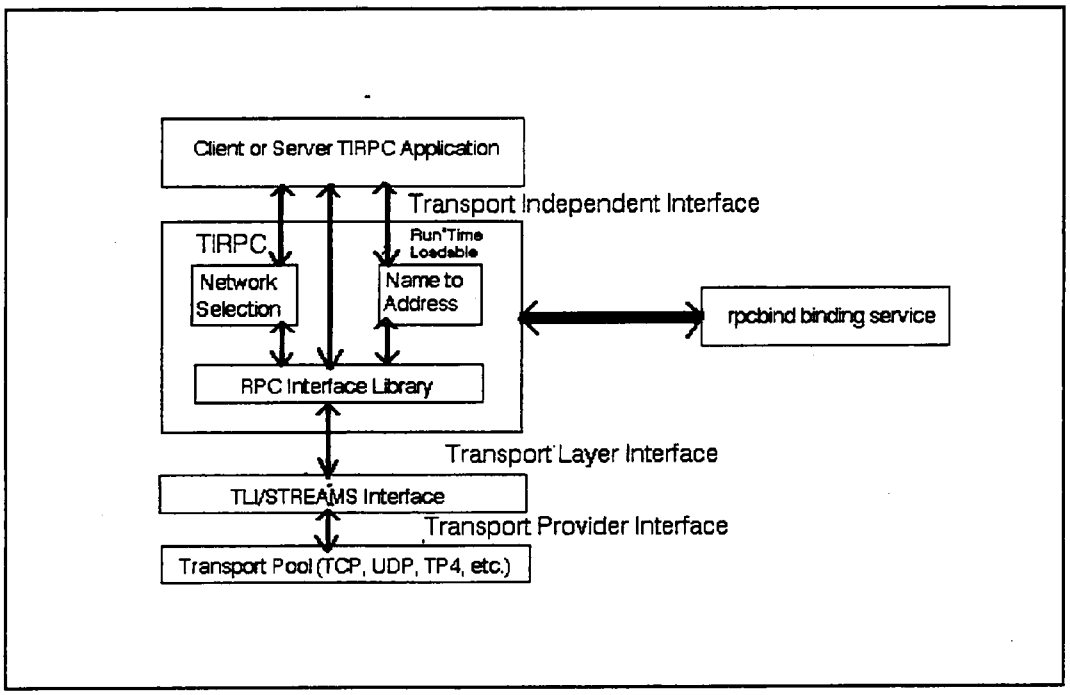

Figure 4.2. **Layers of the TIRPC Interface.**

Adapted from: Bloomer, John. Power Programming with RPC. Sebastopol, California: O'Reilly & Associates, 1992.

A transport-independent RPC system provides a single, consistent programming interface across machines and network transport protocols. The version of RPC that supports transport independence ships with AT&T's System V Release 4 (SVR4). The version separately ported to SunOS as TIRPC is an integral part of Solaris 2.0.

TIRPC is compatible with existing socket-based UDP and TCP versions as long as the code makes no socket calls; it is freely licensed, and the source and its specifications are in the public domain. TIRPC was produced without affecting the existing ONC RPC protocol. ONC RPC applications can therefore

be run as they are and be recompiled as necessary under TIRPC. [4]

ONC enhanced RPC to true transport independence to get the programmer above transport and low-level networking issues. Programmers wanted:

1. **Run-time independence**: Transport selection driven by user preferences and client/server constraints.
2. **Uniform addressing**: Addresses independent of transport, name binding, and resolution mechanisms.
3. **A transport independent API**: Client and server application code not demanding mention of transport. [4]

The RPC interface library used with TIRPC contains many new procedures which generally call the topmost layer of RPC Interface Library and very rarely need to use the Network Selection (NS) library. The Name-to-Address Translation (N2A) modules are almost never called.

Network selection facilitates run-time transport selection. Name-to-address translation provides universal addresses that are independent of the chosen transport. Thus, host names can be mapped to transport addresses. TLI/STREAMS provides the RPC library in a generic way to communicate with all transports (see Chapter 5).

The ***portmapper portmap*** is replaced by ***rpcbind*** to provide a logical (versus IP and port number) interface. ***rpcbind***

registers all RPC service addresses available to the network. RPC services do not need well-known transport addresses specific to a transport. As with ***portmap***, clients query the ***rpcbind*** server on remote hosts to locate the addresses of specified servers. The original RPC protocol, which was always independent of the underlying transport, remains unchanged. [4]

Run-time Transport Independence

The original ONC RPC library supported only UDP and TCP transports. Transports may now be specified by the application, either by class or specific transport, potentially leaving the transport selection available to the user at run-time. A machine's available transports are listed in the **/etc/netconfig** file, and the NETPATH variable may be used to select a transport.

The **/etc/netconfig** file specifies one transport per line, identifying each of the following: netid, type, flags, family, protocol, device address, and loadable module.

A few sample entries are (titles have been added over the columns for clarity):

Netid	Type	Flags	Family	Proto.	Dev. Addr.	Loadable Module
udp	tpi_clts	v	inet	udp	/dev/udp	/usr/lib/tcpip.so
tcp	tpi_cots_or	d	vinet	tcp	/dev/tcp	/usr/lib/tcpip.so
tp4	tpi_cots	v	osi	-	/dev/tp4	/usr/lib/iso.so

Each transport is known by its **netid**. The **type** specifies

whether the transport is connectionless, connection-oriented, or connection-oriented with orderly release; **flags** specifies whether the transport is visible to applications; **family** and **protocol** allow additional transport selection mechanisms; **device address** specifies the full name of the transport device; **loadable module** denotes the name-to-address translation software the RPC library will use to perform name resolution. Multiple modules may be specified to allow multiple name-resolution attempts. [4]

Uniform Addressing

RPCSRC 4.0 used the BSD sockets approach to networking. While sockets are predominately transport-independent, they are fairly nonportable for transport addressing. To develop a truly uniform addressing scheme, ONC could have either added to the sockets interface or chosen the Transport Layer Interface (TLI) as found under AT&T's UNIX. The latter approach was chosen due to existing built-in uniform addressing. BSD 4.4 currently supports OSI.

The concept of well-known ports and IP addresses should all but disappear. Addresses have become string representations of the transport address. Transport-specific characteristics are hidden, since the mechanism for interpreting them is specific to the transport.

Several routines have been added to the TIRPC release but the old interfaces remain for source compatibility reasons. Using TIRPC does not permit using sockets which have been replaced by TLI/STREAMS. While sockets may still be used on the transport, addressing is different, making it difficult to locate socket addresses and names.

Library Levels

One disadvantage of working above the protocol (with TIRPC) is the need to perform more error detection and correction when using an unreliable transport. If the application needs reliable behavior and the transport does not provide that function, the ONC RPC client and server codes must determine the host status on their own. For example, if using UDP, the number and duration of retries may have to be managed by the application; TCP handles many of these things itself. [4]

While the TIRPC API is not that different from the ONC RPC API, there are higher levels of abstraction at which the programmer can work. While programming at a lower level does provide the ability to control the network interface, often resulting in improved performance and functionality, it is more prone to error. It is generally advisable, therefore, that one work at the highest level possible.

The following are levels at which the programmer may develop code:

1. At level one, the characteristics of the transport, the operating system or other low-level implementation mechanisms are not considered. Client and server communication handles are never seen. The client specifies the server and procedure along with the type of transport and receives the results.

2. At level two, one is still not concerned with transports; instead, the user specifies the general class of transport to use. Client and server handles are generated as with the lower-level RPCSRC 4.0 by specifying only the transport type.

3. At level three, the programmer deals with the specifics of the transport. There should be not need to use this level unless there exists a need for more control of clients and servers.

4. Level four consists of a number of procedures for specifying the transport and achieving greater control over it.

5. At level five, one gets full control of the interactions with the ability to control even the smallest details of the transport. [4]

Selection of an RPC System to Use

The most difficult activity in selecting an RPC system to use is that of separating present from future. The benefits of distributed computing appear too great to wait on the sidelines for the market to shake out. The final dominance of one standard or the other may not be clear, especially as both NCS and ONC develop in the same direction. As agreement is reached on one common canonical data representation, common transports with optional transport independence, and the

standardization of network services, NCS and ONC programming will become more logically similar.

Each of the standards currently has enough marketing mass to survive for a while. One viable strategy might be to develop distributed applications under ONC today and to migrate the applications to comply with evolving standards.

ONC RPC, as well as XDR libraries, can be found on machines ranging from DOS PCs to Crays. Many public-domain and officially-supported ports of the ONC/NFS services have been made, including DOS, Macintosh OS, AIX, VM, MVS, VMS, Mach, RTU, PRIMOS, and others.

Sun's ONC RPC does not require the additional purchase of libraries and protocol compilers for each unique machine type, as is the case with RPCTOOL and NCS NIDL. Once packaged with OSF/1, the DECORUM DCE may also be portable, but the final cost has yet to be determined; Sun ONC is publicly available today.

CREATING AN RPC PROGRAM

Creation Procedure

To develop any RPC application, one must follow three steps:

1. Specify the protocol for client/server communication.
2. Develop the client and server programs.
3. Compile the programs.

In defining the protocol, the interface between the client and the server must be defined to establish the framework for application development. If a protocol compiler is used, the names of service procedures and the data types of parameters and return arguments must be identified in the protocol definition. The protocol compiler reads the definition and generates the client and server stubs. [4]

ONC RPC Specifics

The programs designed to use ONC RPCs are programmed in C and the protocol definition language RPCL, which is organized like the C language. An RPCL protocol definition file has an **.x** extension. RPCGEN produces a client stub, a server stub, necessary XDR filters, and a header file to be included by the client and server applications and stubs. A server dispatch function is also generated to take care of validating requests

and of invoking the appropriate service procedures. The service procedures are generally the only code that has to be written for the server.

With RPC, client and server parameters are passed by address. Since they are invoked by the dispatcher, the service procedures must work with pointers. The dispatcher must have an address to a static result in order to encode it and to send the result over the network. Clients must pass addresses to their stubs for the same reason. [4]

In creating an RPC program, the set of remote procedures are grouped into a program which is given a program number, version number, and list of procedures. Each procedure in the list has a procedure number associated with it. The program number defines a set of procedures. The version number permits for the extension of RPC protocols.

When the application on the client machine uses the RPC routines to specify that it wants to communicate with a particular program and version, lower-level RPC routines send a message to the ***rpcbind*** process on the server machine in order to obtain the transport address of the remote program. Upon receipt of the transport address, the client application calls remote procedures within the program. [48]

Since the program number identifies the procedure grouping, it is important that program numbers be unique. To help ensure unique numbers, Sun Microsystems, Inc., keeps track of which RPC programs correspond to which program numbers. A programmer must choose program numbers based on the following criteria:

> 0-1fffffff: This range is reserved for existing RPC programs which are administered by Sun. A number in this range should not be used when creating applications, as it may be assigned to an existing RPC program. A programmer desiring to request an official number must send a description of all associated procedures to the RPC Administrator at Sun Microsystems. If the request is approved, one of these numbers will be assigned to the program.
>
> 20000000-3fffffff: Numbers in this range can be used for site-specific RPC programs. [48]

Each procedure within the program must specify its arguments and return value. Within the RPC framework, each procedure takes one argument and returns one result.

There are three programs that have to be written (see Figure 4.3):

1. A definition of the protocol in the **application.x** file.
2. The service program in **application_svc_proc.c** which contains the actual procedures to be invoked by the server dispatch routine in the compiler-generated stub.
3. The client program **application.c** which must handle attaching to the appropriate service(s) as well as performing other main() functions.

The following file names are foundational to using RPCGEN:

1. Protocol definition file **application.x** is first processed

by RPCGEN.

2. XDR wrapper routines in **application_xdr.c** (filters used by both the client and server).

3. Associated common includes in **application.h**.

4. Client stub in **application_clnt.c**.

5. Server stub in **application_svc.c**. [4]

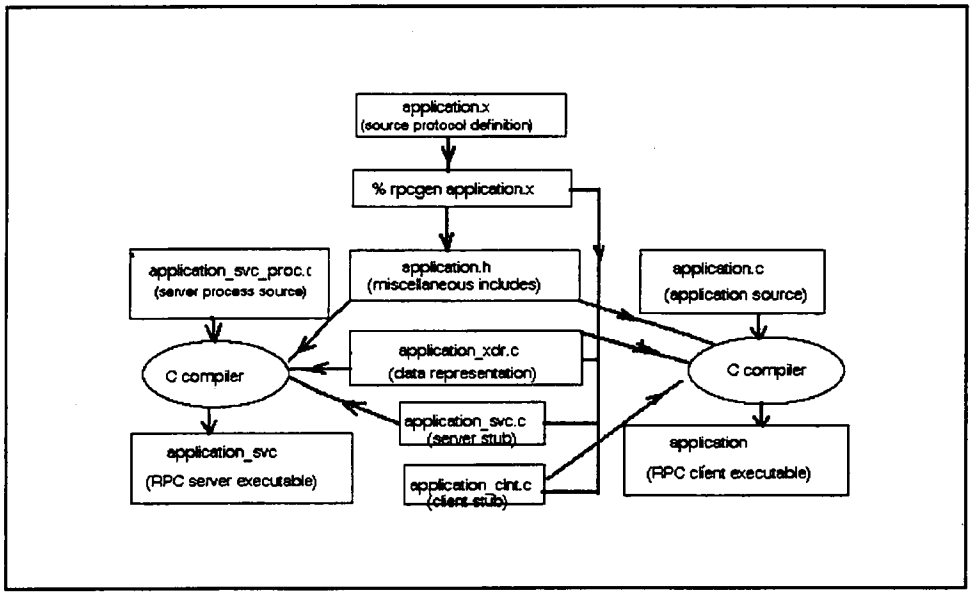

Figure 4.3. **Application Development with RPCGEN.**

Adapted from: Bloomer, John. Power Programming with RPC. Sebastopol, California: O'Reilly & Associates, 1992.

OSF DCE RPC Specifics

In an object-oriented system, all resources in a particular network are characterized as objects. Objects are organized into object types and are manipulated by well-defined operations. An interface is a set of related operations and each object type has one or more interfaces associated with

it. The OSF DCE RPC system permits distributed applications to be defined and built in an object-oriented way (See Figure 4.4). Remote procedure calls are viewed as operations on objects rather than as calls to specific machines or server processes. This approach allows application designers to create programs less dependent on hardware configurations. [78]

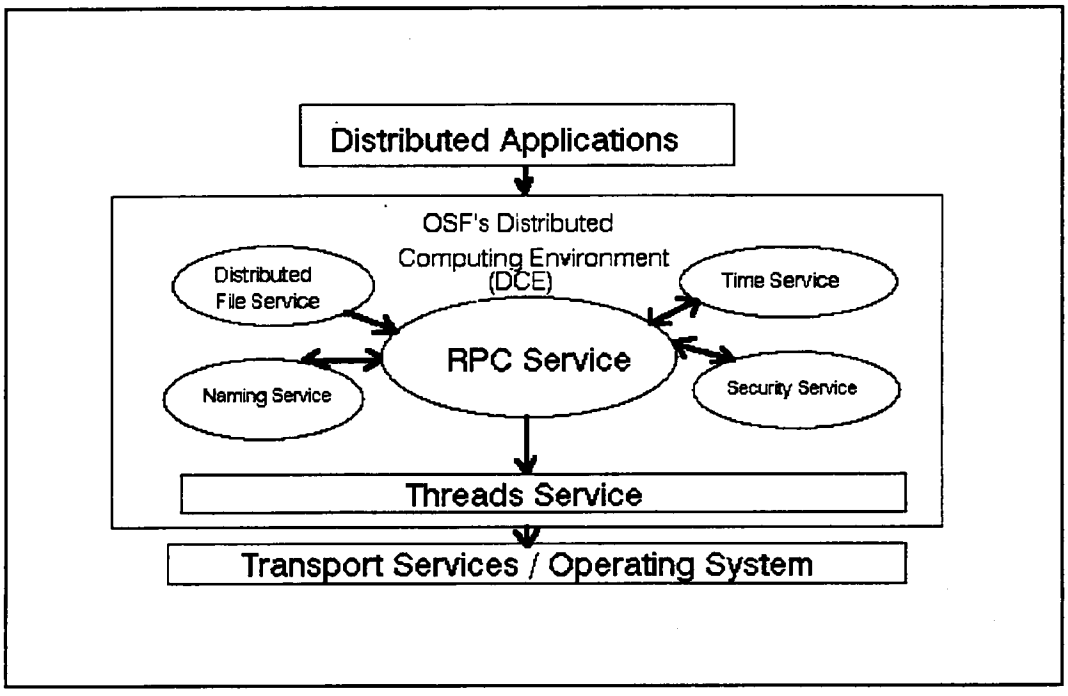

Figure 4.4. **OSF's Distributed Computing Environment.**

Adapted from: Fauth, Dietmar and Shue, Chikong. Remote Procedure Call: Technology, Standardization and OSF's Distributed Computing Environment. Cambridge, MA: Open Software Foundation, 1992.

When a new interface is written, a universal unique identifier (UUID) must be generated using the UUIDGEN utility. The utility generates the number using time and network address

information guaranteed to be unique. An interface UUID enables a client to identify an interface it requires, and it is required in all copies of the interface. The interface is saved in a file with the extension **.idl**.

When the NIDL compiler is invoked, the interface definition goes through two phases: (1) a pre-processing phase that generates the header file and intermediate C language stub files and (2) a compilation phase that generates stub object code.

The following are created from the NIDL compilation: (1) a **C language header file** that contains the definitions needed by the stubs and application code; (2) a **client stub file** to be linked with the client portion of the application; and (3) a **server stub file** to be linked with the server portion of the application (See Figure 4.5). [63]

Four programs need to be written:

1. A definition of the protocol in the **application.idl** file.
2. The service program in **procedure.c** (this name is not absolutely required, but the function is). This file contains the actual procedure(s) to be invoked by the server dispatch routine in the compiler-generated stub.
3. Code to initialize the server (often in a file named **server.c**).
4. The client program **application.c** which must handle attachment to the appropriate service(s) as well the performance of other main() functions. [63]

The following file names are foundational to using NIDL:

1. Protocol definition file **application.idl** is first processed by the NIDL compiler.
2. Associated common includes in NIDL-generated **application.h**.
3. Client stub in **application_cstub.o**.
4. Server stub in **application_sstub.o**. [63]

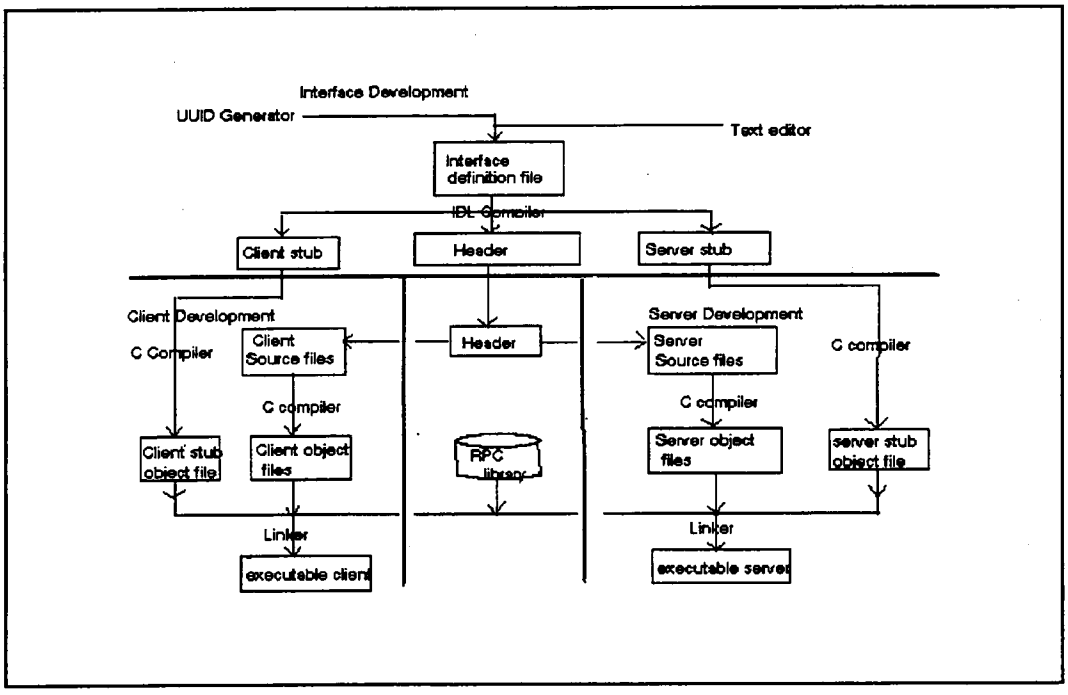

Figure 4.5. **NCS Application Development Tasks.**

Adapted from: Rosenberry, Ward; Kenney, David; Fisher, Gerry. Understanding DCE. Sebastopol, CA: O'Reilly & Associates, Inc, 1992.

After the client and server executables are created, the server program is run to register its interface ID in the database used by the Location Broker and to "activate" the server. A client attempting to connect to the server obtains the server's address from the Location Broker and connects to

the server's "listening" endpoint. Upon successful connection, the server creates a second endpoint, through which all further communication with the client is accomplished.

CHAPTER 5

STREAMS TRANSPORT LAYER INTERFACES

APPLICATION PROGRAMMING INTERFACES AND STREAMS

There are other APIs which might be useful in developing an open computing environment. These APIs utilize the STREAMS framework for communication. The STREAMS framework in general will be discussed before the APIs which employ it are presented.

The STREAMS framework, which replaces the traditional UNIX communications system, reflects the ISO model by taking a layered approach to I/O. UNIX System V Release 4 (SVR4) implements all character I/O devices by way of the STREAMS mechanism. All communication is now STREAMS-based, including terminals, network protocols, and UNIX pipes.

A stream has three parts: the **stream head**, the **device driver**, and **optional modules**. The term downstream is used to refer to the path from the application to the device driver; upstream refers to the path from the device driver to the application (see Figure 5.1).

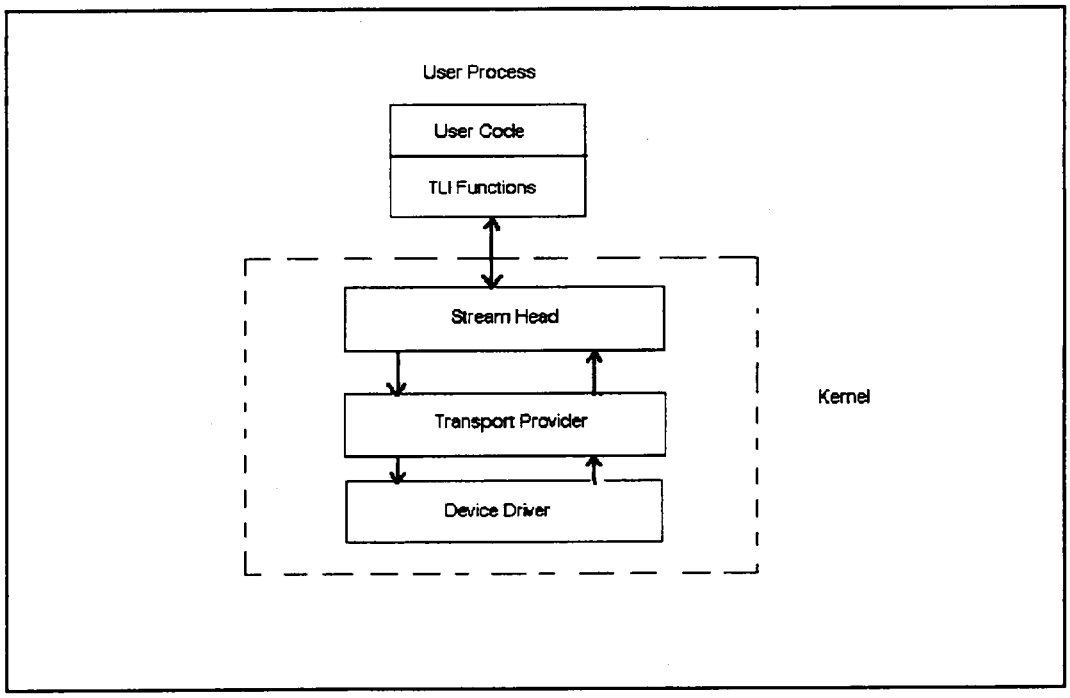

Figure 5.1. **Implementation of TLI.**

Source: Stevens, Richard W. UNIX Network Programming. Englewood Cliffs, New Jersey: Prentice Hall, 1990. Reprinted with permission from Prentice Hall.

The **stream head** is at the top of the stream, and applications interface with it by way of the SVR4 system calls such as **open()**, **close()**, **read()**, **write()**, and **ioctl()**. The stream head interprets the calls and performs the appropriate action. [48]

The STREAMS **modules** are positioned between the stream head and the device driver and perform actions on the data as it travels upstream and downstream.

STREAMS, therefore, provides a full-duplex path between an application and an I/O device. The data path is actually a set of software modules that manipulate data as it travels

between the application and the device driver. An application can customize the data path by choosing which modules should be on the stream. Modules make the STREAMS mechanism powerful in that they can implement the seven layers of the OSI model, encrypt and decrypt data, and perform other functions.

Any number of modules may be pushed onto a stream. Each module gets inserted just below the stream head (a LIFO stack). In networking, the device driver (e.g., Ethernet device driver, token-ring device driver, etc.), becomes the network interface, and all the processing modules between the stream head and the device driver implement a communication protocol (e.g., TCP/IP, OSI, etc). A streams module that accepts data from multiple sources is known as a multiplexer. [72]

The **device driver** is at the bottom of the stream and can be a real device driver, which handles the physical hardware, or a pseudo-device driver (i.e., an emulator of a real device). The device driver accepts messages sent downstream and interfaces with the device. It also processes interrupts from the device and passes messages upstream. [48]

Before the implementation of streams, if UNIX source code were not available and a new feature had to be added to the kernel, a character device driver was required to provide access

through the **open()**, **close()**, **read()**, **write()**, and **ioctl()** system calls.

The UNIX facility is not adequate to handle networking protocols. Also, modules required to implement networking protocols above the device driver do not belong in the device driver. New streams modules can be added in a way similar to adding a new device driver. [72]

THE TRANSPORT LAYER INTERFACE (TLI)

TLI Defined

TLI was introduced with the release 3.0 of UNIX System V in 1986. Prior to this, the focal point of networking in the UNIX community had been BSD releases of UNIX. TLI is a set of user-callable functions which hide the streams interface from the user and provide an interface to the Transport Layer of the OSI model. TLI was implemented using the stream I/O system and is modeled after the ISO Transport Service Definition library of functions. [72]

TLI's direct programming interface to the transport provider (see Figure 5.1) allows one to establish connection, establish communication, and transfer data based on the functionality specified in the ISO Transport Service Specification (ISO

Standard 8072). Using TLI, applications can interface with services that correspond to the Transport Layer of the OSI model.

TLI routines are designed to work with any transport provider. One can use TLI to interface with TCP, UDP, or OSI TP4. TLI provides routines that permit the use of connection-oriented or connectionless services. [48]

Today, the most popular open network protocols are TCP/IP and OSI, but there is no way to predict what will be most prevalent protocols in the future. TLI programming allows a programmer to build applications which run over TCP/IP or OSI today and over future new protocols without re-compiling. TLI's ability to permit an application to work with 32-bit Internet addresses, 15-character X.25 addresses, or variable-length string addresses fosters protocol independence. User data (i.e., user ID) may be sent along with a connection request. Some protocols permit this while others do not. For example, the user ID might be used to accept or reject a connection request. With TLI, one can test whether a protocol has a particular capability and write code applicable to existing capabilities. However, if an application requires any special capabilities, it may be protocol independent, not service independent. With TLI, one can specify service requirements rather than code to meet specific protocol

requirements. [53]

TLI Specifics

In TLI, two communicating processes are called transport endpoints. When discussing TLI, the term "transport user" is used to refer to both the client and server side of an application. Transport users attach themselves to the transport provider through the transport endpoint. [48] The transport provider is a set of routines on a host computer that provide communication support for the user process. TLI is not a transport provider; it is an interface. [72]

All programs that call TLI functions must include the TLI header file **tiuser.h**. The header file includes the definition of a structure called netbuf, which is used for network addresses, protocol options, and user data. Since TLI allows implementations of different transport providers to support different address formats, it defines a generic structure that passes data between TLI functions and user code. The structures are always passed between the user and TLI function by reference (address of the structure passes as an argument to the TLI function). (See Figure 5.2). [53]

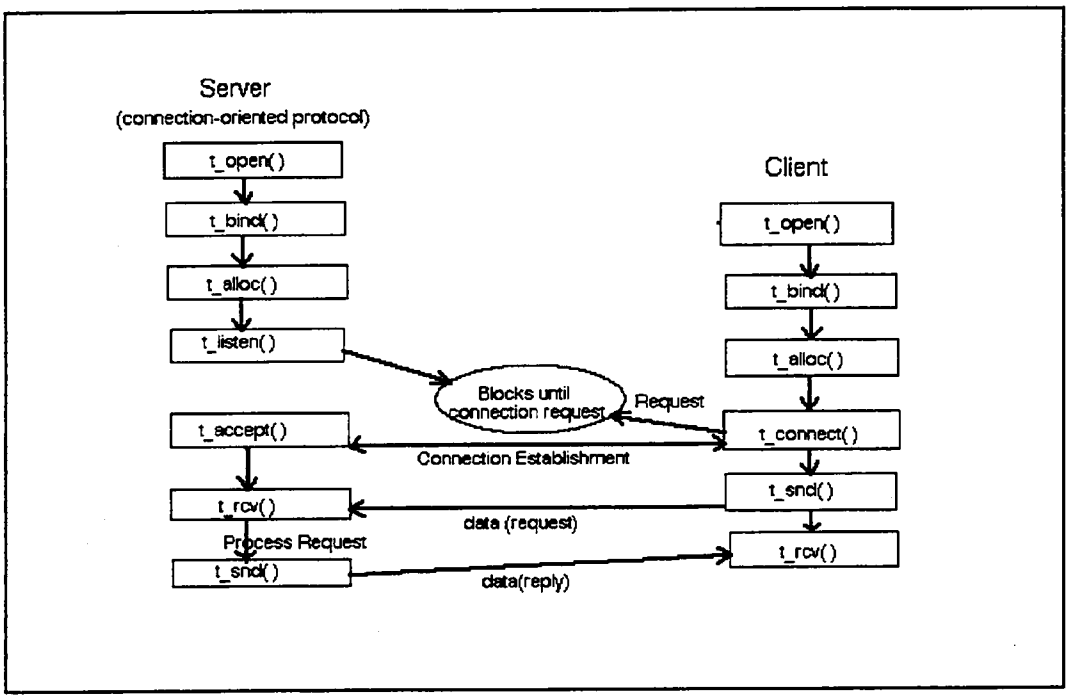

Figure 5.2. **TLI Function Calls** .

Adapted from: Stevens, Richard W. UNIX Network Programming. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

Error Reporting with the t_error() Function

TLI functions set the variable t_errno on error return. The array t_errlist may be indexed by t_errno to obtain an error message for a particular t_errno value. Generally, all TLI function calls return -1 on error and 0 otherwise. The function **t_error()** prints the user's error message string, followed by the appropriate t_errlist value.

Establishing an Endpoint with the t_open() Function

The first step in establishing an endpoint is to open the UNIX file that identifies the specific transport provider. This function returns a file descriptor (a small integer) that is used by other TLI functions.

Using t_alloc() and t_free() Functions

Many of the data structures passed between the user code and TLI functions contain one or more netbuf structures. Each netbuf structure contains a pointer to a buffer used for that particular purpose. The sizes of all these buffers are known once a specific transport provider is selected on the t_open call. To simplify the allocation of these TLI structures, **t_alloc()** is provided. The **t_free()** function frees a structure that was previously allocated by t_alloc. [72]

Naming a Transport Endpoint Using the t_bind() Function

The function **t_bind()** activates the endpoint and must be called for every endpoint, even those established to accept a connection request from a client. The **t_bind()** function assigns an address to the transport endpoint. In all cases, the provider returns the address that it assigns to the endpoint in the return structure. [72]

Connecting to the Transport Endpoint (TEP) Using t_connect()

A connection-oriented client initiates a connection with a server using the **t_connect()** function which can set protocol options, send user data, and return a peer's address in the return t_call structure.

Receiving Connection Requests Using t_listen()

The connection-oriented server waits for a connection request from a client by calling the **t_listen()** function. The t_listen() function blocks and waits for the connection request to arrive. The **t_accept()** function determines whether or not to accept the request. By blocking the **t_listen()** function, TLI separates the listening and accepting functions in way that allows the programmer to refuse a connection request. [53]

Accepting the Connection with the t_accept() Function

Once a connection request has arrived, it may be accepted or rejected. The **t_accept()** function is called to accept the request. The **fd** argument specifies the transport endpoint where the connection request arrived; **connfd** specifies the transport endpoint where the connection is to be established. The original transport endpoint, **fd**, is still available for

listening. [72]

Sending and Receiving with t_snd() and t_rcv() Functions

There are two functions used to send or receive either normal or expedited data: **t_snd()** and **t_rcv()**. The first three arguments in the function are similar to the first three arguments of the standard **read** and **write** system calls. [72]

Inspecting TEPs with the t_look() Function

Every transport endpoint (TEP) has a current event associated with it which can be obtained from the transport provider by calling the **t_look()** function. The integer value returned by this function corresponds to one of the events: T_CONNECT (connection confirmation received); T_DATA (normal data received); T_DISCONNECT (disconnect received); T_ERROR (fatal error indication); T_EXDATA (expedited data received); T_LISTEN (connection indication received); T_ORDREL (orderly release indication); T_UDERR (datagram error indication). [72]

Orderly Connection Release

TLI supports abortive and orderly release. Abortive release does not guarantee delivery of outstanding data, while orderly release does guarantee this. As has been stated, all transport

providers provide abortive release; orderly release is optional. A process sends an orderly release indication by calling the **t_sndrel()** function; the release is accepted by calling the **t_rcvrel()** function. The transport provider notifies the user process by returning an error on the next TLI function call. [72]

Abortive Connection Release

The function **t_snddis()** is used for abortively disconnecting an established connection and rejecting a connection request. The function **t_rcvdis()** is used to receive the disconnect request.

Detaching the Transport Endpoint with the t_unbind() Function

The **t_unbind()** routine detaches the transport endpoint from its address. It may be used on both the client and server sides of an application.

Closing the Connection with the t_close() Function

The **t_close()** function allows a transport provider to free any local resources that it has allocated for a specific endpoint. The **close()** system call is also called for this file descriptor. Other actions accomplished by this function depend

on two conditions: (1) if the calling process is the last one with an open endpoint and (2) whether or not the endpoint is connected. If the transport endpoint is connected when the function is called, and if it is the last process to reference this endpoint, the connection is broken with an abortive disconnect. If another process has the endpoint open, a disconnect is not sent to the other end. [72]

X/OPEN TRANSPORT LAYER INTERFACE (XTI)

XTI Defined

X/Open is an independent, world-wide, open systems organization supported by most of the world's largest information systems suppliers, user organizations, and software companies. X/Open's goal is to bring users greater value from computing through the practical implementation of open systems. To accomplish this goal, they have combined existing and emerging standards into a comprehensive system known as the Common Applications Environment (CAE). This environment covers standards above the hardware level needed to support open systems. It provides for portability and interoperability of applications and allows users to move between systems with minimal retraining. The components of CAE are defined in the X/Open CAE Specifications, which contain definitions of practical Application Programming Interfaces

(APIs) to enhance portability of applications at the source code level and definitions of protocols and protocol profiles which enhance the interoperability of applications. A marking of "XPG" on a vendor's product indicates conformance to CAE specifications. [8]

Because of the significance of TLI for Network programming, TLI (with minor modification) has been included in the X/Open CAE (Common Applications Environment). The interface specification is called XTI (X/Open Transport Interface). [74]

XTI is not dependent upon a particular version of UNIX and is primarily concerned with the ISO Transport Service Definition; it may be adopted for use over other transport providers. [8]

XTI is based on the SVID issue 2, volume III, Networking Services Extensions, and provides a refinement of the TLI where necessary. This refinement provides additional commentary or explanatory text where TLI is either unclear or lacking in detail and provides modifications to the interface. These modifications have been minimized to avoid any fundamental changes to the interface defined by TLI.

XTI recommends a subset of the functions and facilities defined in TLI and makes modifications to these functions and/or facilities where necessary. An application written in

conformance to XTI might not be immediately portable to a provider written in conformance to TLI. XTI does not address the management aspects of the interface; neither does it address how addressing must be done so that the application is truly portable. [8]

XTI Specifics

The following are features of XTI which are extensions to System V Release 3 version of TLI: (1) some functions may return more error types; (2) the transport provider identifier has been generalized to remove the dependence on a device driver implementation; (3) additional events have been defined to help applications make full use of the asynchronous features of the interface; (4) additional features have been introduced to t_snd(), t_sndrel(), and t_rcvrel() to allow fuller use of transport providers; (5) to increase application portability, options have been defined for certain types of transport service; (6) because most XTI functions require read/write access to the transport provider, the usage of flags O_RDONLY and O_WONLY has been withdrawn; and (7) XTI checks the value of qlen and prevents an application from waiting forever after issuing a t_listen().

In XTI, a common subset of functions in connection-oriented and connectionless services are always implemented: t_bind(),

t_close(), t_look(), t_open(), t_sync(), t_unbind().

If a connection-oriented transport service is provided, t_accept(), t_connect(), t_listen(), t_rcv(), t_rcvconnect(), t_rcvdis(), t_snd(), and t_snddis() are supported.

If XTI supports access to connectionless transport services, the following functions are always implemented: t_rcvudata(), t_rcvuderr(), and t_endudata(). Since this document primarily discusses the connection-oriented transport services, no further mention will be made of these connectionless functions.

Utility functions supported by XTI are: t_alloc(), t_free(), t_error(), t_getprotaddr(), t_getinfo(), t_getstate(), t_optmgmt(), t_strerror().

The orderly release mechanism is supported only by T_COTS_ORD type providers, and use with other providers will cause an error. Creating applications using an orderly release is not recommended if an application using TCP must be portable to the ISO Transport Layer.

Optional mechanisms provide the ability to manage more than one incoming connect indication at any one time. The address of the caller is passed with the **t_accept()** function call and

may be optionally checked by the XTI implementation.

Transport Endpoints (TEPs), as in TLI, specify the communication path between the transport user and a specific transport provider (identified by the local file descriptor [**fd**]). When a transport provider identifier is opened, a local file descriptor is opened to identify the transport endpoint. A transport provider is a transport protocol that provides the services of the Transport Layer. All requests to the transport provider must pass through the TEP. A TEP can support only one established transport connection at a time. To be active, the TEP must have an address assigned to it using the **t_bind()** function. A transport connection is characterized by the association of two active endpoints. To guarantee portability, the only operations which applications may perform on any **fd** returned by **t_open()** are those defined by XTI and fcntl(), dup(), or dup2(). Other permitted operations have system-dependent results.

The identifier parameter of the transport provider that is passed to the **t_open()** function determines the required transport provider. For portability, the identifier parameter should not be hard-coded into the application source code. For an application to manage multiple transport providers, it must call **t_open()** for each provider. If a server is waiting for connection indications from several transport providers, a TEP

must be opened for each provider.

One process can open several **fds**. In synchronous mode, the different actions of the transport connections must be managed sequentially. Several processes may share the same **fd** but must be synchronized so as not to issue a function unsuitable to the current state of the endpoint. The transport provider treats all users of the TEP as a single user, and multiple processes using the same endpoint must coordinate their activities so as not to violate the state of the provider.

Several endpoints can be bound to the same protocol address, but only one at a time may be listening for incoming connections; others may be in a data transfer state or may be establishing a transport connection as initiators.

The transport service interface supports two modes of service: connection mode and connectionless mode. One TEP may not support both modes simultaneously.

Two levels of error handling are defined for a transport interface. The first is the **library error** level. Failures are indicated by a return value of -1. The second level of error is at the **operating system service routine** level. A special library error number has been defined which is generated by each library function when the operating system service

routine fails or some other general error occurs.

There are two modes for handling events: **synchronous** and **asynchronous**. The transport service interface is inherently asynchronous; several events may occur independently of the actions of the transport user. In the default synchronous mode, the transport primitives wait for specific events before returning control to the user. Therefore, the user cannot perform any tasks while waiting. The asynchronous mode provides a mechanism for notifying a user of an event without forcing the user to wait for an event.

A process that issues functions in synchronous mode must still be able to recognize some asynchronous events and act on them. The special transport error [TLOOK] handles this situation. The t_look() function is invoked to identify that a specific event has occurred when an error is returned. Polling is another means to notify a process that an asynchronous event has occurred.

All events that occur at a TEP are stored by XTI and are retrievable one at a time by the **t_look()** function. An event is outstanding on a TEP until it is consumed and every event has a corresponding consuming function for the event. [8]

As stated before, XTI supports two modes for handling events:

synchronous (blocking) mode and **asynchronous** (non-blocking) mode. The mode is set by using the O_NONBLOCK flag passed as parameter to **t_open()**. Synchronous mode is the default mode of execution and is good for applications that need to wait for events to occur or maintain only a single transport connection. If an application needs to maintain more then one transport connection, asynchronous mode is better, because it does not force the user to wait for events at a TEP. If the user wants to receive data using the **t_rcv()** function in asynchronous mode and data are not available, **t_rcv()** returns a failure code to the user immediately. The user may keep polling for incoming data by repeating **t_rcv()** or use **t_look()** to check for outstanding events. Otherwise, if the user wants to receive data using the synchronous mode and no data are available, it can call block until either the data, a disconnection indication, or interrupt signal have arrived.

The following functions are mandatory for the connection establishment phase and have the same function as in TLI: t_connect(), t_rcvconnect(), t_listen(), and t_accept().

Once the transport connection is established, data can be sent or received. Two modes of data transmission are supported: record-oriented and stream-oriented. Message boundaries are not preserved in stream-oriented, but they are preserved in record-oriented. The TLI-related functions t_snd() and

t_rcv() are mandatory for the data transfer phase.

XTI supports both destructive disconnect and orderly release, but support for orderly release is optional.

If multiple processes use the same endpoint, synchronization is required among processes so that the state of the TEP remains consistent. A single process in UNIX may execute several different programs by making use of the **exec** system call. The **t_sync()** function helps handle **execs**. If a process **execs** another process that will handle a transport endpoint, then the TLI data structures that were part of the initial process disappear. The TLI functions in the spawned process do not know anything about the original process' endpoint; the endpoint is valid and the file descriptor is passed from the first process to the second.

The second process must call the **t_sync()** function, which queries the transport provider in the kernel for information about the endpoint and initializes the appropriate TLI structures. The second process may then use the file descriptor with TLI functions appropriate for that endpoint. It is not necessary to call t_sync() after a fork, since the new process has a copy of the data area of the calling process. [72]

Writing Protocol-Independent Software Using XTI

An application programmer who writes XTI programs faces two important portability challenges: portability across protocol profiles and portability across different platforms. Sometimes, options are inseparably coupled with a definite protocol or protocol profile and using them lowers portability across protocol profiles. Different vendors might offer products with different option support, and vendors will always implement subsets that suit their needs. Every application of a protocol profile accessible by XTI can be used with the default values of options, and applications can thus can be written without implementing options. Any application program processing options retrieved from an XTI function should discard all options unknown to it in order to make it less system platform dependent. [8]

The following is true of a portable application written using the XTI interface.

1. The application should use only the mandatory functions and mechanisms of XTI.

2. The concept of message boundaries may not be supported by all transport providers, and assumptions should not be made regarding the preservation of logical data boundaries across a connection.

3. If an application is not intended to be run over an ISO transport provider, the name of the device should not be hard-coded into the application.

4. Protocol-specific service limits returned on t_open() and t_getinfo() functions must not be exceeded. Limits must be adhered to throughout the communication process.

5. The user program should not look at or change options specific to an underlying protocol. The t_optmgmt() function enables the user to access default protocol options from the transport provider; the options may or may not be passed as an argument on appropriate connection establishment functions.

6. Protocol-specific addressing issues should be hidden from the user program. The application must access destination addresses in an invisible manner (i.e through a name server).

7. Reason codes associated with t_rcvdis() are protocol-dependent. This information should not be interpreted if protocol independence is a concern.

8. Error codes associated with t_rcvuderr() are protocol-dependent. The user should not interpret this information if protocol independence is a concern.

9. The optional orderly release facility of the connection-mode service should not be used by programs targeted for multi-protocol environments. This facility is not supported by all connection-based transport protocols, and its use will prevent programs implementing it from communicating with ISO open systems.

10. The semantics of expedited data are different across different transport providers. An application should avoid using expedited data calls if it is to run over different transport providers. [8]

DETERMINING WHEN TO USE A STREAMS TRANSPORT LAYER INTERFACE

TLI or XTI can and should be used to create sophisticated applications that have full control over the transport provider. Naturally, which interface is supported in which

environment determines which one can be used. Applications using TLI/XTI can negotiate transport-provider options, manage multiple network connections simultaneously, and monitor transport-provider events.

The interface may be used when creating applications that cannot use RPCs. RPC is easier to use than TLI, because it calls most of the lower-level networking routines; but some networking applications do not fit the RPC model. For example, a remote-login procedure which must maintain two-way communication between a client and server machine could not be written using RPCs. The client side of an application must send commands, data, and signals to the server side of the application. The server side of the application must send command results and error information to the client side. All of these required activities make this type of application unsuitable for the remote procedure call (RPC) model. [48]

One important consideration in whether to select the RPC, TLI, or XTI interface is whether data conversion will be required between the systems on which the client and server code resides. TLI and XTI do not support data conversion and any conversion required would have to be programmed into each application.

CHAPTER 6

API COMPARISON

INTRODUCTION

An investigation was made into the similarities and differences in using the RPC and TLI APIs. An assessment of the use of the APIs in the current computing environment was made and conclusions drawn as to possible usefulness of both (or similar) APIs in the future.

Two separate initiatives were undertaken: (1) performance testing using the ONC RPC and TLI interfaces, and (2) a subjective analysis as to the actual use of these APIs and the DCE RPC API in the programming environment.

The goal of the performance testing was to determine what impact on the network, timing constraints, and total throughput would be evidenced by using code written to perform exactly the same task in both RPC and TLI.

PERFORMANCE ANALYSIS

Method of Study

Performance analysis applications were developed using both the TLI and ONC RPC APIs. Three different Sun machines, all mapped to the same file system, were used. The applications consisted of a server program on machine 3 ; a client program on machine 2 ; and a shell script on machine 1 (see Appendix A). See APPENDICES B and C for RPC and TLI source code listings.

The public-domain **tcpdump** packet-capture software was used to capture packet data for the analysis.

A shell script was designed to: (1) begin the ***tcpdump*** program to capture packet traffic data; (2) delete the ***tcpdump*** process upon completion of the transfer of each block of data; (3) set the block size for transfer and call the client program using that number; (4) produce 15 files of data by performing data transfers of increasing block sizes; (5) produce a count of the total number of packets in each transfer; (6) write the block count and total packet number to the output file; and (7) copy the file of packet data relating only to the transfer to a file denoted by sample number and block size.

The client software used the block size passed to it from the script file to request that amount of data to be transferred back to it. The system time was captured before and after the call and the total time for the transfer (to the user) was calculated and written to the common output file for that sample.

The server software accepted the size of the block, multiplied that number by 1024 (to obtain the number of bytes for transfer), and transferred the requested block of data. Block numbering started at 1 and proceeded by doubling the previous number, up to 128. This numbering scheme was chosen to provide a clear picture of transfer trends while employing only a few data points.

The transfers did not employ disk I/O, since all data transferred was obtained from an array held in the server computer's memory.

The server code for each test filled an array with the amount of data required by the client. While this action is not required to transmit test RPC data (due to the fact that data may be any arbitrary sequence of bytes), it is required for the TLI code to function properly and was left in both the TLI and RPC server code for consistency. Several samples were taken to determine the possible impact of filling the array.

Since the tests were to determine the total impact of performing the same function, and since the array filling time was a constant to both servers being tested, this amount of time is not deemed excessive. This is a constant measurement which has a smaller impact on the total results as the amount of data actually being transferred increases.

Sampling was done late at night to help ensure a quiet net. Before the transfers were done, a check was made to see that data was not flowing between the machines on which the client and server software was located. The packet transfer numbers over all samples were compared, and outlying high and low were values discarded. Ten of the most closely-aligned samples from each block transfer of the original 15 were averaged to provide the final data (See table 6.1).

A similar type of testing was completed to determine the impact of the connection establishment setup and breakdown which would be reflected in the total packet count and time statistics. The program code for the TLI and RPC tests was altered to give information which did not include file transfer and data array filling time. The request information containing the block size was left in the TLI code, since that is considered overhead not necessary in RPC and similar data content was included in the originating RPC client call. Ten of the most consistent values for each sample were averaged.

TABLE 6.1

TLI AND RPC PERFORMANCE TEST DATA

AVERAGE OF 10 SAMPLES

DATA POINT	BLOCK (KB)	# PKTS	TIME (MICROSEC)
	TLI DATA		
1	1	10	162710
2	2	11	166513
3	4	13	166971
4	8	17	1168365
5	16	25	1175609
6	32	39	1189163
7	64	66	1221321
8	128	124	1707606
	RPC DATA		
1	1	17	236862
2	2	21	238004
3	4	29	248586
4	8	45	281500
5	16	77	208013
6	32	143	359675
7	64	284	1179944
8	128	582	3810553

Results of the Study

The fact that the activity on the test machines was minimal at the time of all tests adds reasonableness to their results. It was found that to fill the array with one byte and receive the time back took 445.7 microseconds (0.0004457 seconds). To fill the array with 131072 bytes (128 KB) took 59415 microseconds (0.059415 seconds). A subsequent test revealed that it took 34 microseconds (0.000034 seconds) just to retrieve the time information.

The average connection establishment/destruction information for TLI was: (1) 8 packets transferred and (2) 28185 microseconds (0.028185 seconds). Comparable information for RPC was: (1) 9 packets and (2) 44937 microseconds (0.044937 seconds). An analysis of the packet data showed seven zero data-length TCP packets used in the connection establishment /destruction in both TLI and RPC. One packet of the TLI data represents the overhead incurred by having to send the block size as a separate call (as opposed to sending it with the call, as in RPC). RPC packet data contained two UDP packets at the beginning of the setup. The result that the RPC connection establishment/destruction took roughly twice as much time as that for TLI was unexpected.

The number of RPC packets required to complete the task was

much greater at larger message sizes (See Figure 6.1). Analysis of the RPC packet data on larger transfers revealed that at most 3 data packets (often 2) were sent to the client before a zero-data acknowledgement (ACK) packet was sent from the client to the server. A spot check of several files on RPC packet data revealed no instance where all three packets preceding an ACK (from client to server) contained more than 4356 bytes of data (in two packets of 1460 bytes of data and one packet of 1436 bytes of data). The majority of the three-packet blocks before an ACK contained 4000 bytes of data (two 1460 data packets and one 1080 byte data packet). The TLI packet data revealed that many packets of data would be sent before any communication was made from client to server. ACKs were randomly found in the packet information, but were not seen nearly as often as in the RPC transfer information. The TLI packet data contained many sequences of 1460-byte data packets. The total number of ACK packets and incomplete filling of data packets affected the average of the data bits/packet (see Figure 6.2).

It is interesting to note that little difference in time to complete the application was found up to a block size of 4 KB. The bytes/second and total time comparisons are very similar (see Figures 6.3 and 6.4). At block sizes from 8 KB to 32 KB, the TLI software began to take a longer time to perform the required task than did the RPC software. At 64 KB, the two

APIs seem to again synchronize in time, but at 128 KB the TLI software took the lead. It can only be assumed that the significant amount of packet transfer required by RPC to perform the function caused the time delay. One might conjecture that some form of windowing or buffering mechanism is being employed by the client and server stubs to allow time for data conversion and prevent data overrun. This activity would generate a significant amount of communication-type packets (i.e. ACKs).

Conclusions from the Study

For smaller message sizes, the time required to perform the function and the impact on the network did not vary significantly between the APIs. At larger message sizes, however, TLI might be the choice over RPC, since fewer packets are used to transfer the required data and the transfer is accomplished in less time. It is important to note the impact of the RPC application on the network. If a programmer elected to use RPCs for large message transfer in a heavily-used application, the increased packet load could become significant. Naturally, performance is not the only factor to consider in selecting an API for use; difficulty of development using an API, availability of the API on various platforms, and the pool of available programming expertise are also other valid factors to consider.

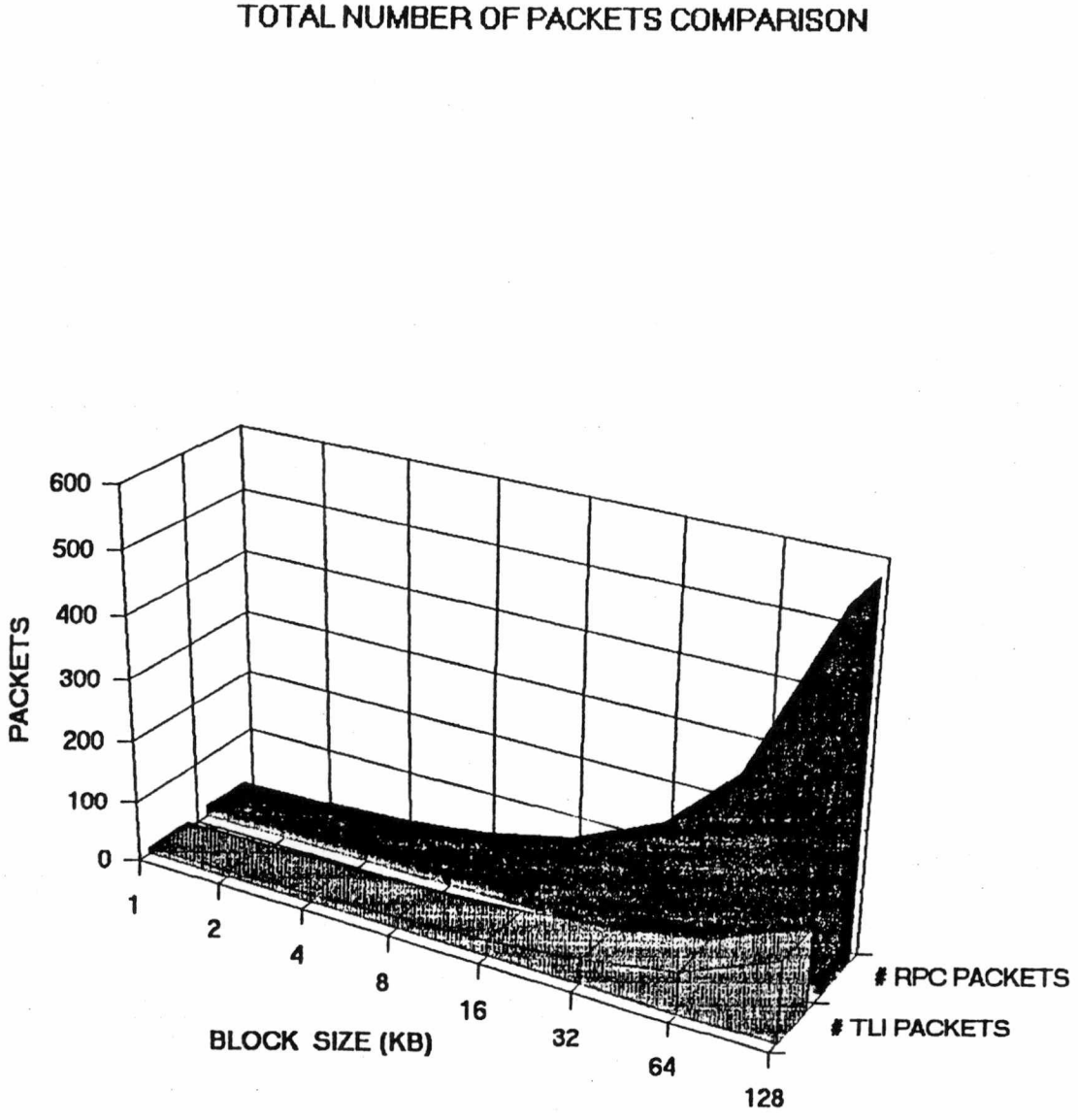

Figure 6.1. **Total Number of Packets Comparison**

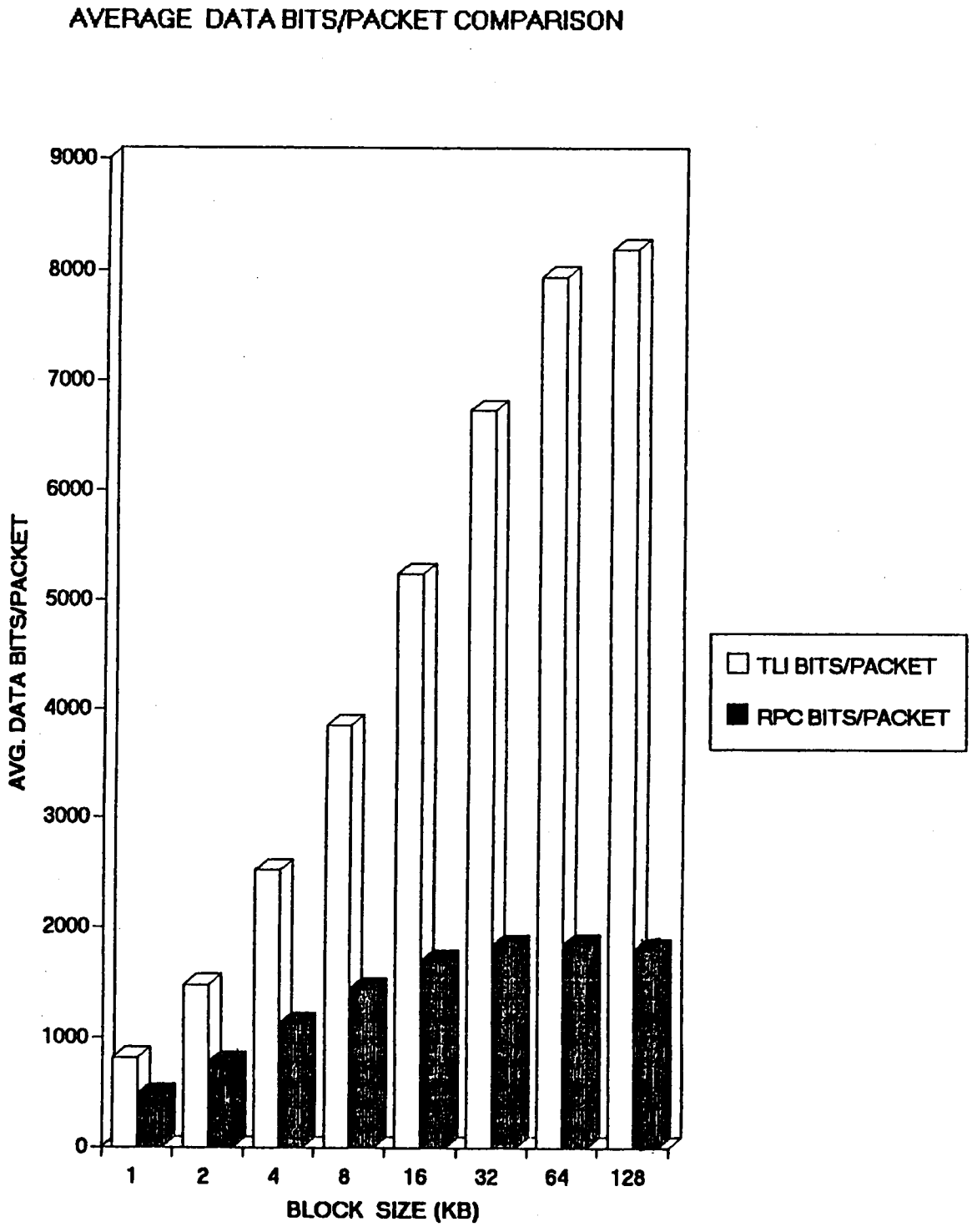

Figure 6.2. **Average Data Bits/Packet Comparison**

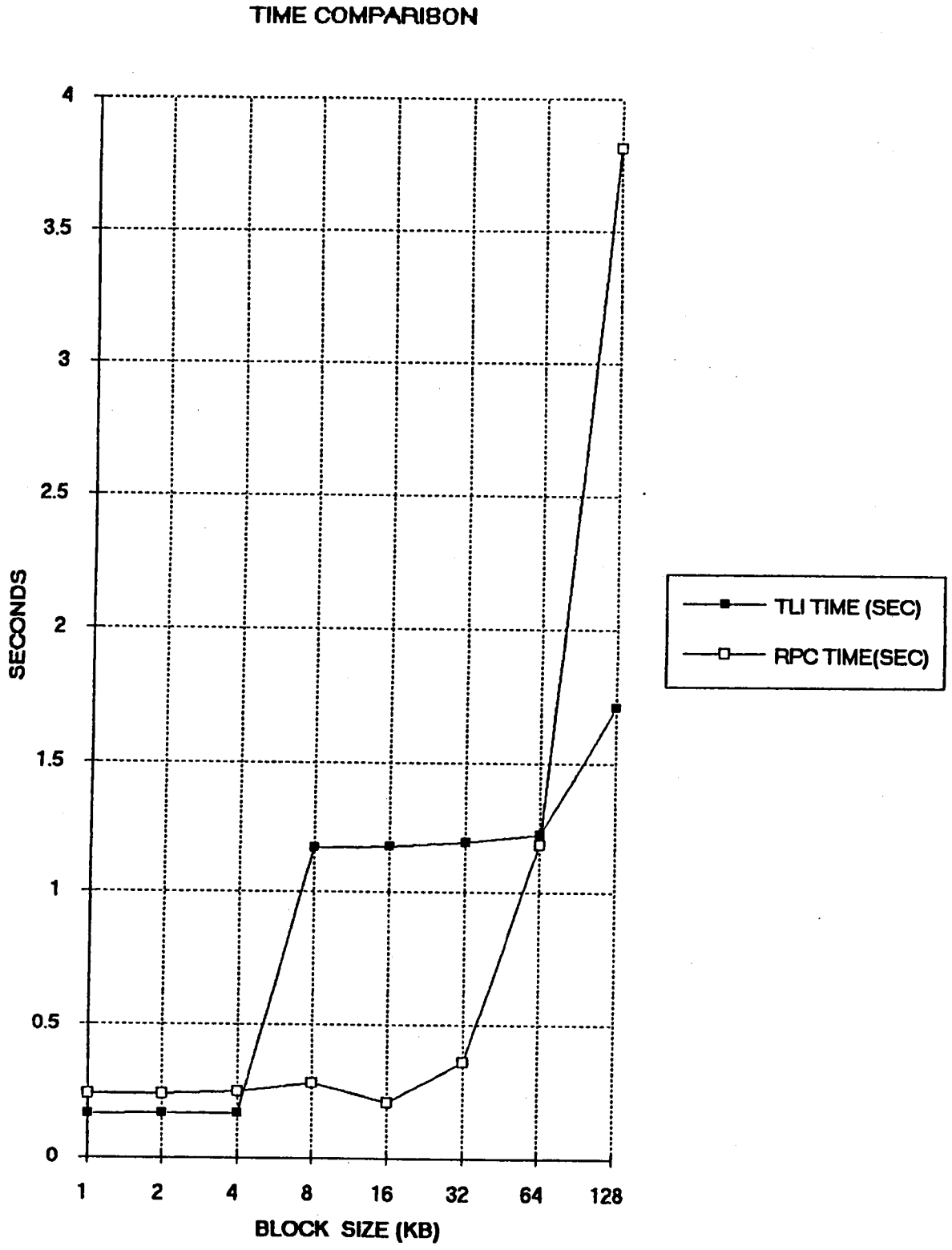

Figure 6.3. **Time Comparison**

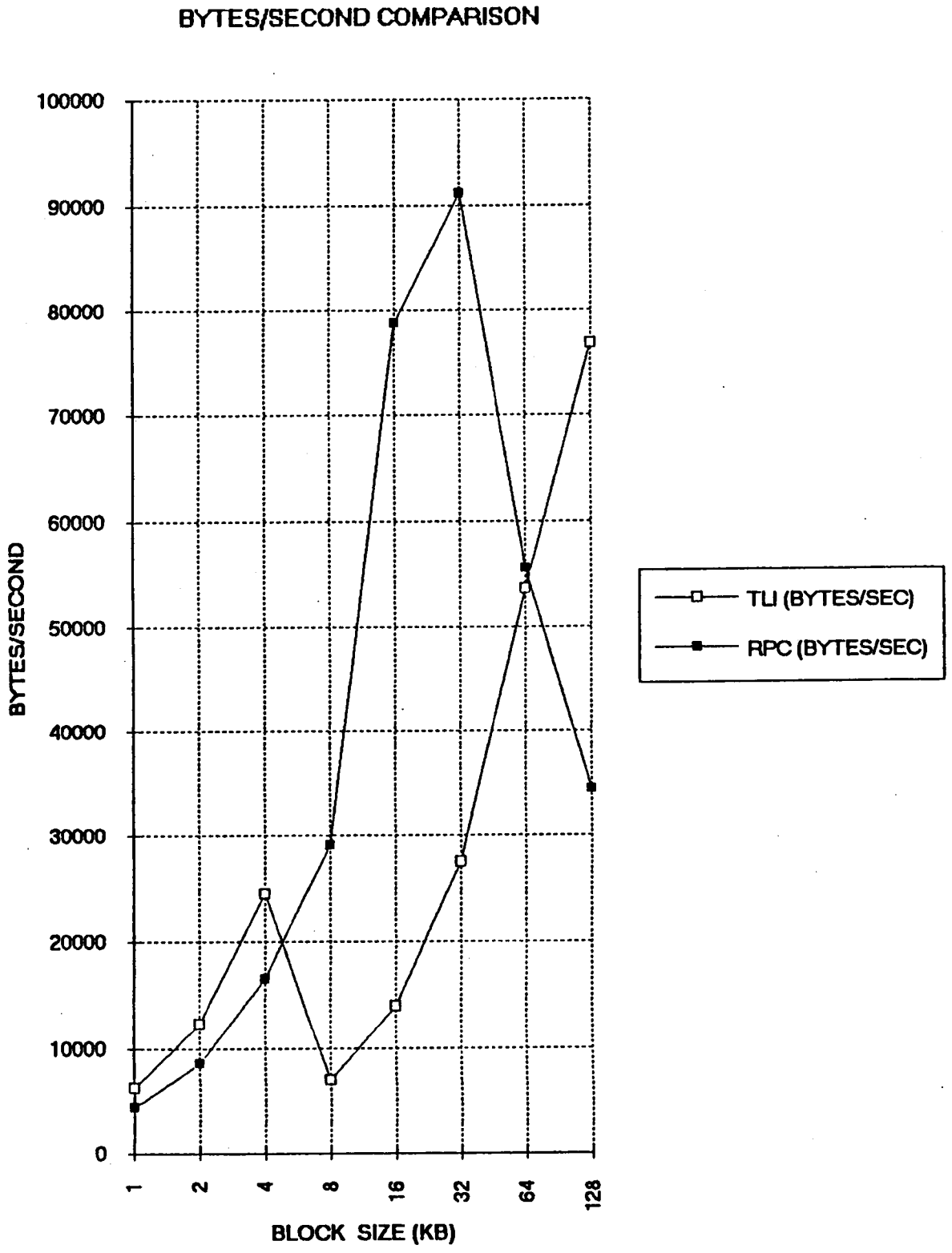

Figure 6.4. **Bytes/Second Comparison**

APPLICATION DEVELOPMENT ANALYSIS

Method of the Study

This study has more subjective elements than the previous study. False starts and pitfalls are mentioned to provide a basis from which the reader may deduce some of the difficulties currently involved in programming in an open system environment.

Two Sun machines and two IBM RS/6000 machines were used to provide different computer platforms for the testing both the RPC and TLI program development. A third IBM RS/6000 machine was used to assess RPC program development in the DCE environment. Since the DCE software was not available on any other machine to which the author had access, use of NCS RPCs was limited to a single machine.

A program was developed which provided information on TLI services available on UNIX-based machines. The program was based on an example in Stevens book [72] (see APPENDIX D for source code listing). The software was run on the IBM and Sun machines. The following listing shows the results of those runs. The references to TOSITP0, TOSITP2, and TOSITP4 are for TP0, TP2, and TP4 respectively. Information was actually printed for these transport protocols when the OSI software

was operational on the IBM machines.

```
IBM Machine

          name  addr    options tsdu    etsdu   connect discon  servtype
 //dev/xti/tcp  16      0       0       1024    -2      -2      2 COTS_ORD
 //dev/xti/udp  16      -2      8192    -2      -2      -2      3 CLTS
TOSITP0: system error, No such file or directory
TOSITP2: system error, No such file or directory
TOSITP4: system error, No such file or directory

Sun Machine

          name  addr    options tsdu    etsdu   connect discon  servtype
      /dev/tcp  16      0       0       1       -2      -2      1 COTS
/dev/udp: System error: No such file or directory
TOSITP0: System error: No such file or directory
TOSITP2: System error: No such file or directory
TOSITP4: System error: No such file or directory
```

Initially, it was thought that a client/server application program could be developed on the IBM machines to run over both TCP and TP4. This application would communicate between the two similarly-configured IBM machines. In investigating this possibility, it was found that the OSI software on the machines used software that was compatible with the XTI specification. A minimal time-server client/server application was developed, but it was found that the XTI software did not support TCP. When the OSI software was installed on the IBM machines, a version of TLI was also installed as a side benefit of that software installation. TLI software is not automatically installed on the IBM systems when the operating system is installed. A manual page on XTI was available on the machine; one on TLI was not.

A time-server program was developed on one IBM computer to use TLI in order that communication could be accomplished freely

with the other IBM machine and with one of the Sun machines. When IBM technical personnel were asked why there were two separate pieces of software (one for TCP/IP and one for OSI), no solid reason was offered. According to these personnel, however, XTI will support TCP in the future. To allow the IBM machines to be used for more pressing purposes and, since development of software on what was essentially proprietary standalone software seemed counter to the purpose of the study, an administrative decision was made to remove the OSI software from the IBM machines.

Use of freely-available Sun software (TIRPC) in order to achieve transport-level independence was investigated. The TIRPC software was obtained for installation on the two Sun machines. It was necessary to purchase additional software (U.S. Encryption kits specific to the operating system on each machine), since the TIRPC software could not be installed without this software. The system administrator installed the software on one of the machines, but a query of users of the system revealed that a piece of software that used RPCs had been installed on the system subsequent to the purchase of the Encryption Kits, making installation of the TIRPC software on that system unwise. The two machines were selected because both OSI and TCP/IP software were available on both machines. Using the TIRPC software to communicate using either TCP or TP4 was the original intent. Since the software could not be

installed on the second target machine, this objective could not be realized. It was also found that TIRPC supported only TCP. The software which will perform the same function in the Sun Solaris operating system will support TP4. Using RPCs over TLI over the network without having the TIRPC software on both machines in the client/server pair was not possible since the transport calls are entirely different.

The Sun machine on which the bulk of the ONC RPC programming was done used the TIRPC software. It employed ***rpcbind*** rather than ***portmap*** to register the service addresses available. There was no problem in porting developed RPC code from that machine to one that used ***portmap***.

Each file-transfer application developed provided a server program which was started in the background to wait for a connection from a client. When the client connection was established, the client provided the name of the server, the name of the file to be transferred, and the name of the file to which the information should be transferred.

Results of the Study

Installation of RPC Code

Development of the ONC RPC program went fairly well (see APPENDIX E for source code listings). Documentation for RPC development is fairly sparse and difficult to obtain. Developing the protocol file first and then using the protocol compiler-generated variables took a while to master, as did learning to coordinate de-bugging of the program between the user-developed protocol specification file (TRANS.X), the client program (TRANS.C), the server program (TRANS_SVC_PROC.C), and (where necessary) the protocol-compiler-generated files.

Although there were very slight differences in the code generated by the ***rpcgen*** compiler on the Sun and IBM machines, the differences were not significant enough to warrant inclusion of the IBM ***rpcgen***-generated code in this document.

Development of the DCE RPC code was significantly more difficult (See APPENDIX G for source code listing). Previous development of ONC RPC code was extremely helpful in providing a basic understanding of RPC action. While there are fewer C files generated in the DCE RPC application development process, development of a server-initialization program was

required; this program was significantly more involved than that required in ONC RPC server development. Access-right set-up to the name service so that the application could be registered in the run-time library was fairly involved and required that the DCE administrator make several set-up attempts. The applications would not run without the proper setup of the name service. The method of transferring information from the server was by reference, using either pointers or arrays.

Installation of the TLI Code

The TLI program was also fairly difficult to develop (see APPENDIX F for source code listings). The TLI software on the Sun machine had to be configured specifically in the kernel of the machine, making it necessary to ask the system administrator to configure the machine properly. Program code generated by the RPC compiler for the establishment of network connections had to be user-generated in TLI. Since an orderly release was not supported by the TLI software on the test machines, a pause had to be added after the server sent a block of data to allow correct receipt of the file by the client. If this were not done, the server sent all the data in rapid succession, and the client would be unable to process it. With TLI, user data cannot accompany a connection request (as it can in RPC), so a separate **t_snd** call had to be used to

send the name of the file for transfer from the source machine. The TLI software requires the IP address of the server (without some mapping by the programmer); the RPC software can take the server name or IP address.

Attempting to use string-read calls from the source file and sending that string over the network produced a large number of small packets. Small packets do not utilize the network very efficiently. While strings might have been read and artificially placed into a longer string, design for this study seemed best for explanatory purposes if character read and write in the client and server were employed.

The TLI software on the Sun machine was far easier to use and was much less prone to inexplicable errors than was the IBM software. TLI on the IBM required ***strload*** to be run by someone with root privileges before TLI would work properly. The command line for compiling on the IBM is:

cc -o example example.c -ltli

On the Sun the command line is:

cc -o example example.c -lnsl

CONCLUSIONS

Open Systems

When reading trade literature one would perceive that software transport independence is a present reality. It is, but not in the terms one would think. Transport independence today often means simply the ability to select either UDP or TCP. The slow acceptance of OSI protocols has made development of software that will support those protocols lag behind. The writer's personal opinion is that the OSI protocols as they are today may never be widely used. Having worked with OSI software and particularly having attempted to define addresses within the "free-form" OSI address structure is partially responsible for this opinion. The addressing structure, while allowing extreme flexibility, also makes standardization difficult, if not impossible. Incompatibility between address structures of different versions of OSI software on the same machine type make maintenance and communication a real problem.

Even if the TIRPC software had been usable and a client/server application had been developed using that software, the application would not have been portable to other computing platforms (as RPC programs using ONC RPC are today). The objective of developing less transport-dependent software is

a good one, but it appears that the most widely-accepted specification moving in that direction today is the Distributed Computing Environment (DCE) from the Open Software Foundation (OSF). It seems logical that the only way to achieve true "open systems" using transport-independent communication is to use some widely-accepted standard such as DCE for program development. However, the DCE environment is still in the developmental and early deployment stages, making current widespread application development in DCE not only impractical, but impossible.

Vendor Motivation

What a designer generally sees in the development of an "open" computing environment is that vendors (understandably) walk a fine line between (1) being the first in the field with a marketed product that may or may not be acceptable (and which has cost a great deal in research and development) and (2) being too late into the arena with a product that is acceptable, thus losing out on revenue. One must face the fact that vendors are in business to make money and not necessarily to make consumers' lives easier. If making life easier makes money, then they go for it, but not until profit actually looms on the horizon. Even when developing products according to a standard, vendors must use proprietary solutions to fill in the "holes" where standards are not fully

defined, thus making their products incompatible with other vendors, who have also been forced to implement proprietary solutions. Vendors must have an edge in the marketplace; so they "add value" to the standards, resulting in products that are incompatible with those of other vendors. This is the foundation of a capitalistic culture; to mandate absolute conformance to product development standards without deviation begins to smack of totalitarianism.

Network Loads

One thing that is vitally necessary in the future is some way for application developers to determine what impact their code is having on the network. Personal investigation into the use of X-Windows software in a distributed computing environment showed that the programmer is generally unaware of network usage (nor do they often care unless the program bogs down).

Often an application could be made much more efficient if a means of analyzing the network load generated by the application were more readily accessible to the programmer. It should not be necessary for an application developer to acquire a protocol analyzer, learn how to use it, learn how to interpret the data received, and then make programming decisions. Protocol analyzers and general access to the networks are usually not available to the general public

because of the cost of the equipment and concern for security respectively. Protocol analysis software on machines is generally available only to system administrators.

It would be extremely helpful if a programmer could start some software on a particular machine and determine the network load between that machine and another specified machine that is being generated by a selected application. It should not be necessary for individual packets to be inspected; this would help alleviate security concerns.

Development of a method to assess the impact of an application on a network would be a very viable future research topic. This research and developmental activity should include the design of a simple way for application programmers to retrieve and utilize that information.

Future Network Environment

The vast Internet exists today primarily because the U.S. government mandated that all government computers be able to communicate with each other. That is the intent of the GOSIP specification for government computers: unfortunately, the impetus to create a standardized communication mechanism does not exist, primarily due to the current existence and success of the TCP/IP protocol suite. Perhaps the only thing that will

really usher in a new era of standardized computing is for the TCP/IP protocol suite to become either totally obsolete, requiring implementation of a completely new brand of network computing, or for TCP/IP to be totally accepted as the standard. TCP/IP extensions might be added that would provide additional functionality for the TCP/IP suite making it a more serious contender as a long-term internetworking protocol. Most assuredly, improved physical communications media and gigabit computing standards will help pave the way to the establishment of a networking environment in which only the more robust network protocols will survive.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. AIX Version 3 for RISC System/6000 OSI Messaging and Filing/6000: User's and System Administrator's Guide. Palo Alto, California: International Business Machines Corporation, 1991.

2. Black, Uyless. TCP/IP and Related Protocols. New York, New York: McGraw-Hill Book Company, 1992.

3. Black, Uyless. OSI: A Model for Computer Communications Standards. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

4. Bloomer, John. Power Programming with RPC. Sebastopol, California: O'Reilly & Associates, 1992.

5. Braden, R., ed. "Requirements for Internet Hosts - Communication Layers". Request for Comments: 1122. Oct. 1989.

6. Brambert, D. ed. Guide to Internetworking. San Francisco: Miller Freemen Inc., 1993.

7. Brown, Wendy and Colin M. Simpson. The OSI Dictionary of Acronyms. New York, New York: McGraw-Hill, Inc., 1993.

8. CAE Specification: X/Open Transport Interface (XTI). Berkshire, United Kingdom: X/Open Company Limited, 1992.

9. Cole, Gerald D. Implementing OSI Networks. New York, New York: John Wiley & Sons, Inc., 1990.

10. Comer, Douglas E. Internetworking with TCP/IP: Volume I: Principles, Protocols, and Architecture. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

11. Dickson, Gary and Alan Lloyd. Open Systems Interconnection. Sydney, Australia: Prentice Hall, 1992.

12. Edwards, Morris. "Who or what will jolt life into open systems?". Communications News Oct. 1992: p. 47(1).

13. Fauth, Dietmar and Shue, Chikong. Remote Procedure Call: Technology, Standardization and OSF's Distributed Computing Environment. Cambridge, MA: Open Software Foundation, 1992.

14. Fetterolf, Peter. "Connectivity: the sum of its parts: the structure of modern networks". Byte Nov. 1991: p. 197(8).

15. Fiedler, David. "Networking UNIX". Byte June 1991: p. 363(3).

16. Gray, Pamela A. Open Systems: A Business Strategy for the 1990s. London, England: McGraw-Hill Book Company, 1991.

17. Halsall, Fred. Data Communications, Computer Networks, and Open Systems. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1992.

18. Hardcastle-Kille, S.E. "Encoding Network Addresses to Support Operation over Non-OSI Lower Layers". Request for Comments: 1277. Nov. 1991.

19. Haywood, Peter. "What about the I in OSI? Vendors try to ease users' pains in getting OSI products to interwork". Data Communications Nov. 1989.: p. 56(3).

20. Henshall, John and Sandy Shaw. OSI Explained: End-to-End Computer Communication Standards. West Sussex, England: Ellis Horwood Limited, 1990.

21. Hindin, Eric. M. "A common API for UNIX; the first standard API for any OSI layer will simplify the coexistence and migration of applications". Data Communications Mar. 1991: p. 60(1).

22. Hindin, Eric. "New LAN protocol promises multi-megabits throughput: would replace 'bottleneck' protocols like TCP-IP, TP4; IBM has jumped on the bandwagon". Data Communications Mar. 1989: p. 49(4).

23. Hindin, Eric M. "Getting real about distributed, multivendor applications". Data Communications Dec. 1991.: p. 94(2).

24. Horwitt, Elisabeth. "Pressured COS overhauls: users frustrated by poor OSI, TCP/IP interoperability." Computerworld 7 Dec. 1992: p. 1(2).

25. Horwitt, Elizabeth. "Straddling the OSI fence." Computerworld 15 July 1991: p. 42(2).

26. Horwitt, Elizabeth. "I want my TCP-IP". Computerworld 31 July 1989: p. 45(2).

27. I'Anson, Colin and Adrain Pell. Understanding OSI Applications. Englewood Cliffs, New Jersey: Prentice Hall, 1993.

28. "Interoperability: Cornerstone of Open Systems." IDC White Paper. Cambridge, MA: Open Software Foundation, 1993.

29. Juneau, Lucie. "The trials, tribulations and triumphs of TCP/IP; despite compatibility snags and implementation glitches, users say TCP/IP is worth it". Computerworld 7 Oct. 1991: p. 103(4).

30. Knightson, K.G., T. Knowles, and J. Larmouth. Standards for Open Systems Interconnection. New York, New York: McGraw-Hill Book Company, 1988.

31. Koolwine, Art. "To dream an interoperable dream".Telephony 20 March 1989: p. 34(2).

32. Lang, R. and R. Wright R., eds. "A Catalog of Available X.500 Applications". Request for Comments: 1192. Jan. 1992.

33. Livingston, Dennis. "Here comes OSI...sort of". Systems Integration Sept. 1991: p. 43(4).

34. MacKinnon, Dennis, William McCrum, and Donald Sheppard. An Introduction to Open Systems Interconnection. Oxford, England: W. H. Freeman and Company, 1990.

35. Malamud, Carl. "Streams helps comm-based services go with the flow". Data Communications May 1990: p. 129(5).

36. Malamud, Carl. "Paying by the inch for OSI documentation". Data Communications June 1990: p. 37(2).

37. McClain, Gary R. Open Systems Interconnection Handbook. New York, New York: McGraw-Hill Book Company, 1991.

38. Meijer, Anton and Paul Peeters. Computer Network Architectures. Rockville, Maryland: Computer Science Press, 1983.

39. Miller, Mark A. Internetworking: A Guide to Network Communications. San Mateo, California: M&T Books, 1991.

40. Miller, Mark A. Troubleshooting TCP/IP: Analyzing the Protocols of the Internet. San Mateo, California: M&T Books, 1992.

41. Morgan, H. Larry and S Bob Sheikh. "TCP/IP-based data transport services for LAN/WAN interconnection". Telecommunications Jan. 1992: p. S15(3).

42. Muller, Gunter and Robert Blanc, eds. Networking in Open Systems. Berlin, Germany: Springer-Verlag, 1987.

43. Nance, Barry. "Interoperability today; the OSI stack provides a blueprint for interoperability and shows that our reach still exceeds our grasp". Byte Nov. 1991: p. 187(7).

44. Naugle, Matthew. Network Protocol handbook. New York: McGraw-Hill, Inc., 1994.

45. "Open Systems for Technical Staff: Unit 8: Interoperability (Part I)." Naperville, Illinois: International Training Group, 1990.

46. "Open Systems for Technical Staff: Unit 9: Interoperability (Part II)." Naperville, Illinois: International Training Group, 1990.

47. Padocano, Michael. "Open the door to non-unix systems with Telnet". Systems Integration Jan. 1992: p. 27(1).

48. Padovano, Michael. Networking Applications on UNIX System V Release 4. Englewood Cliffs, New Jersey: Prentice Hall, 1993.

49. Padovano, Michael. "Get greater connectivity by putting OSI protocols into UNIX". Systems Integration Oct. 1991: p. 33(1).

50. Padovano, Michael. "To win federal business, you need to know GOSIP". Systems Integration Nov. 1991: p. 37(1).

51. Piscitello, David and Chapin, A. Lyman. Open Systems Networking. Reading, Massachusetts: Addison-Wesley Publishing Company, 1993.

52. Rago, Stephen A. UNIX System V Network Programming. Reading, Massachusetts: Addison-Wesley.,1993.

53. Reiken, Bill and Weiman, Lyle. Adventures in UNIX Network Applications Programming. New York, New York: John Wiley & Sons, Inc., 1992.

54. Remote Procedure Call in a Distributed Computing Environment. White paper. Cambridge, MA: Open Software Foundation, 1990.

55. Reynolds, J. and J. Postel. "Assigned Numbers". Request for Comments: 1060. Mar. 1990.

56. Rose, Marshall. The Open Book: A Practical Perspective on OSI. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

57. Rose, Marshall T. and Dwight E. Cass. "ISO Transport Services on Top of TCP". Request for Comments: 1006. May 1987.

58. Rose, Marshall. "Making the transition from TCP/IP to OSI". Computerworld 7 Oct. 1991: p. 99(1).

59. Rosenberry, Ward; Kenney, David; Fisher, Gerry. Understanding DCE. Sebastopol, CA: O'Reilly & Associates, Inc, 1992.

60. Russell, D. The Principles of Computer Networking. Cambridge, England: Cambridge University Press, 1989.

61. Schnaidt, Patricia. Enterprise-Wide Networking. Carmel, Indiana: SAMS publishing, 1992.

62. Semich, William J. "The distributed connection: DCE". Datamation 1 Aug. 1991: p. 28(3).

63. Shirley, John. Guide to Writing DCE Applications. Sebastopol, CA: O'Reilly & Associates, Inc, 1992.

64. Simpson, David. "5 routes to coexistence: Open Systems Interconnection". Systems Integration April 1990: p. 82(5).

65. Simpson, David. "The streams machine". Data Communications May 1990: p. 62(6).

66. Snell, Ned. "Why TCP/IP still has the edge". Datamation 1 Oct. 1992: p. 38(4).

67. Spohn, Darren. L. Data Network Design. San Francisco: McGraw-Hill, Inc., 1993.

68. Stallings, William. Data and Computer Communications. New York, New York: Macmillan Publishing Company, 1988.

69. Stallings, William. Networking Standards: A Guide to OSI, ISDN, LAN, and MAN Standards. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1993.

70. Stallings, William. "On moving to OSI". Byte Sept. 1990: p. 265(1).

71. Stevens, Richard. UNIX Network Programming notes. San Francisco, California: INTEROP conference, 1992.

72. Stevens, Richard W. UNIX Network Programming. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

73. Sunshine, Carl A. Computer Network Architectures and Protocols. New York, New York: Plenum Press, 1989.

74. Tang, Adrian and Sophia Scoggins. Open Networking with OSI. Englewood Cliffs, New Jersey: Prentice Hall, 1992.

75. Tannenbaum, Andrew S. Computer Networks. Englewood Cliffs, New Jersey: Prentice Hall, 1988.

76. Unix Press. Network Programming Interfaces: Unix SVR4.2. Elglewood Cliffs, New Jersey: Prentice Hall: 1992.

77. Wexler, Joanie M. "Open systems rivals move to link standards". Computerworld 23 Sept. 1991: p. 1(2).

78. X/Open Specification: Networking Services, Issue 3. Berkshire, United Kingdom: X/Open Company Limited, 1992.

APPENDICES

APPENDIX A

SHELL SCRIPT FOR PERFORMANCE TEST

```
#############################################################
# Script file used in performance testing of both TLI and RPC
# Actual machine names have been replaced by numbered machine
# references for clarity and security
#
# The script iteratively passes requested block parameters to
# the client program, traps packet information, writes
# pertinent data to the output file, and deletes old files
#############################################################
# Initialize the counter for number of samples
@ times = 1
# Start sampling
while ($times < 16 )
rm output < y
cp blank output
# Send the block values for this sample
@ block = 1
while ($block < 129 )
# Remove old files; y is a file containing the value y
rm test1 < y
rm test2 < y
rm timefile < y
rm hold < y
rm stopit < y
rm killit < y
# start tcpdump - packet capturing software
tcpdump -q > test1 &
# Write the block number to output file
echo $block >> output
# Make certain there are enough packets in the sample
sleep 20
# Call client here with value (this is RPC call)
# TLI call was: rsh machine2 tliclnt machine3 $block
rsh machine2 echoc machine3 $block
# The following lines provide more "padding" packets
sleep 20
rsh machine3 cat holdit > /dev/null
sleep 20
# Write out the process value into the file hold
ps -ax | grep -e 'tcpdump' > hold
# Run a program called stripit to pull out only process number
# See program listing at script end
stripit
# Paste file named kill which contains the words kill -9 to
# process id file
paste kill stopit > killit
# Make killit executable
chmod +x killit
# Remove the tcpdump process
killit
# Filter the packet file for the appropriate values
# First filter creates a file of only specified packets
grep -e 'machine2' test1 | grep -e 'machine3' > test2
# Second filter puts the count of the packets into output
grep -c -e 'machine2' test2 >> output
# File containing packets copied to unique file
cp test2 $times.$block
# Put the time from the client into output file
cat ~/timefile >> output
# Increment the block size
@ block *= 2
end
# Save the results of this sample to unique file
cp output $times
@ times = $times + 1
```

```
end


#############################################################
# The file stripit.c which produces a file containing
# only the process ID of the tcpdump process
#############################################################

##include <stdio.h>

#main()

#{
#     FILE *g, *f;
#     int c,i, timethru;

#     g = fopen("hold","r");
#     f = fopen("stopit","w");
#     c = getc(g);
#     timethru = 1;
# Write out characters until the space after process id
#     while ( c != EOF)
#     { if ((c == 32) && (timethru != 1)) exit(1);
#       if ((c != 32) && (timethru == 1)) putc(c,f);
#      if (timethru != 1) putc(c,f);
#      c = getc(g);
#     timethru++;
#    }
#fclose(g);
#fclose(f);
#}
```

APPENDIX B

RPC PERFORMANCE TESTING - SOURCE CODE

```
/*****************************************
 * FILE echoc.x - protocol definition file *
 *****************************************/

typedef string block<20>;

typedef string message<>;

struct file {
 char data<>;
};

program ECHOPROG {
     version ECHOVERS {
          file GETDATA(block) = 1;
     } = 1;
}= 0x20000020;

################################################################

/*********************************************************
 * FILE - echoc_svc_proc.c - server code for testing RPC   *
 * performance                                             *
 *********************************************************/

#include <rpc/rpc.h>
#include "echoc.h"
#include <stdio.h>

#include <string.h>

       int c,i ; /* Useful variables */
       file *
       getdata_1(blocksize)
           block *blocksize;

       {      static file openfile;
              char hold[140000];
              char *msg;

              c = 0;
              i = atoi(*blocksize);
       while (c != (i * 1024)) /* fill string with data */
       {      hold[c] = 'Z';
              c++;}
       hold[c] = '\0';
       msg = hold;
       c = strlen(msg);
       openfile.data.data_val = msg;
       openfile.data.data_len = c;
      return(&openfile);
}/*

################################################################

/*********************************************************
 * FILE - echoc.c - code used in performance testing of RPC *
 *********************************************************/

#include <stdio.h>
#include <rpc/rpc.h>
#include "echoc.h"
#include <sys/types.h>
#include <sys/time.h>
```

```
main(argc, argv)

int argc;
char *argv[];

{       FILE *g; /* File pointer for time file */
        CLIENT *cl; /* pointer for connection */
        char  *server; /* server name */
        file *result; /* pointer for returned result */
        long start, finish; /* variables - start and end times */
        struct timeval before, after; /* for time values */
        struct timezone tz;

        server = argv[1];
        gettimeofday(&before, &tz); /* Get start time */
        cl = clnt_create(server, ECHOPROG, ECHOVERS, "tcp");
        if (cl == NULL) { clnt_pcreateerror(server);
                          exit(1);
                        }
        result = getdata_1(&argv[2],cl); /* make the call */
         if (result == NULL)
        clnt_perror(cl,server);
        gettimeofday(&after, &tz); /* Get ending time */
        start = (before.tv_sec * 1000000) +  before.tv_usec;
        finish = (after.tv_sec * 1000000) + after.tv_usec;
        g = fopen("timefile","w"); /* Write the time to file */
        if (g == NULL)
        {       printf("Cannot open the timefile  \n");
                exit(1);
        }
fprintf(g, "%ld \n", finish - start);
fclose(g);
exit(0);

################################################################

 /*********************************************************
 * FILE - echoc.h                                         *
 * Please do not edit this file.                          *
 * It was generated using rpcgen.                         *
  *********************************************************/

#include <rpc/types.h>


typedef char *block;
bool_t xdr_block();

typedef char *message;
bool_t xdr_message();

struct file {
        struct {
                u_int data_len;
                char *data_val;
        } data;
};
typedef struct file file;
bool_t xdr_file();

#define ECHOPROG ((u_long)0x20000020)
#define ECHOVERS ((u_long)1)
#define GETDATA ((u_long)1)
extern file *getdata_1();
```

```
#############################################################

/*************************************************************
/* FILE - echoc_svc.c                                        *
 * Please do not edit this file.                             *
 * It was generated using rpcgen.                            *
 *************************************************************/

#include <stdio.h>
#include <rpc/rpc.h>
#include "echoc.h"

static void echoprog_1();

main()
{
        register SVCXPRT *transp;

        (void) pmap_unset(ECHOPROG, ECHOVERS);

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL) {
                fprintf(stderr, "cannot create udp service.");
                exit(1);
        }
        if (!svc_register(transp, ECHOPROG, ECHOVERS, echoprog_1, IPPROTO_UDP))
        {
                fprintf(stderr, "unable to register (ECHOPROG, ECHOVERS, udp).");
                exit(1);
        }

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL) {
                fprintf(stderr, "cannot create tcp service.");
                exit(1);
        }
        if (!svc_register(transp, ECHOPROG, ECHOVERS, echoprog_1, IPPROTO_TCP))
        {
                fprintf(stderr, "unable to register (ECHOPROG, ECHOVERS, tcp).");
                exit(1);
        }

        svc_run();
        fprintf(stderr, "svc_run returned");
        exit(1);
        /* NOT REACHED */
}

static void
echoprog_1(rqstp, transp)
        struct svc_req *rqstp;
        register SVCXPRT *transp;
{
        union {
                block getdata_1_arg;
        } argument;
        char *result;
        bool_t (*xdr_argument)(), (*xdr_result)();
        char *(*local)();

        switch (rqstp->rq_proc) {
        case NULLPROC:
                (void) svc_sendreply(transp, xdr_void, (char *)NULL);
                return;
```

```
        case GETDATA:
                xdr_argument = xdr_block;
                xdr_result = xdr_file;
                local = (char *(*)()) getdata_1;
                break;

        default:
                svcerr_noproc(transp);
                return;
        }
        bzero((char *)&argument, sizeof(argument));
        if (!svc_getargs(transp, xdr_argument, &argument)) {
                svcerr_decode(transp);
                return;
        }
        result = (*local)(&argument, rqstp);
        if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
                svcerr_systemerr(transp);
        }
        if (!svc_freeargs(transp, xdr_argument, &argument)) {
                fprintf(stderr, "unable to free arguments");
                exit(1);
        }
        return;
}

###############################################################

/*************************************************************
 * FILE - echoc_clnt.c                                       *
 * Please do not edit this file.                             *
 * It was generated using rpcgen.                            *
 *************************************************************/

#include <rpc/rpc.h>
#include "echoc.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

file *
getdata_1(argp, clnt)
        block *argp;
        CLIENT *clnt;
{
        static file res;

        bzero((char *)&res, sizeof(res));
        if (clnt_call(clnt, GETDATA, xdr_block, argp, xdr_file, &res, TIMEOUT)
        != RPC_SUCCESS) {
                return (NULL);
        }
        return (&res);
}

###############################################################

/*************************************************************
 * FILE - echoc.xdr.c                                        *
 * Please do not edit this file.                             *
 * It was generated using rpcgen.                            *
 *************************************************************/

#include <rpc/rpc.h>
#include "echoc.h"
```

```
bool_t
xdr_block(xdrs, objp)
        XDR *xdrs;
        block *objp;
{
        if (!xdr_string(xdrs, objp, 20)) {
                return (FALSE);
        }
        return (TRUE);
}

bool_t
xdr_message(xdrs, objp)
        XDR *xdrs;
        message *objp;
{
        if (!xdr_string(xdrs, objp, ~0)) {
                return (FALSE);
        }
        return (TRUE);
}

bool_t
xdr_file(xdrs, objp)
        XDR *xdrs;
        file *objp;
{
        if    (!xdr_array(xdrs,    (char    **)&objp->data.data_val,    (u_int
*)&objp->data.data_len, ~0, sizeof(char), xdr_char)) {
                return (FALSE);
        }
        return (TRUE);
}
```

APPENDIX C

TLI PERFORMANCE TESTING - SOURCE CODE

```
/*********************************************************
 *  FILE - tlisrvr.c - TLI code for performance testing *
 *********************************************************/

extern int t_errno ;     /*  TLI library errorcode          */
#include "header.h" /*  TLI definitions    */
#define MAX 140000
#include <string.h>

main( argc, argv )  /*  arg count, arg vector          */
char *argv[];

{      /*  transport endpoints, receive flag     */
int fd,newfd,b,flags,event,i,c;

struct sockaddr_in client, server; /* Internet addresses */

struct t_bind    req;      /* TLI struct for bind req    */
struct t_call    *callptr;  /* TLI struct for call req    */
char buffer[10];
char   DATA[MAX];     /* data buffer area      */
while(1)
      {if ((fd=t_open(DEV_TCP,O_RDWR,(struct t_info *)0)) < 0)
     {  t_error( argv[0] );  exit(1); }

       memset( &server, 0, sizeof(server)); /* Server size */
       server.sin_family = AF_INET;          /* Internet domain  */
       server.sin_port   = htons(TCP_PORT);  /* Advertised port  */
       server.sin_addr.s_addr = htonl(INADDR_ANY); /* any board   */

       /* req.addr.maxlen = maximum bytes buf can hold */
       /* req.addr.buf = buffer containing address */
       /* req.addr.len = size of the address */
       /* req.qlen = maximum connections to support */
       req.addr.maxlen  =  req.addr.len  =  sizeof( server );
       req.addr.buf     = (char *) &server ;  req.qlen  =  5;

       /* Bind to the "listening endpoint */
       if ( t_bind ( fd, &req, (struct t_bind *) 0 ) < 0 )
       { t_error( argv[0] );  exit(2); }

       callptr = (struct t_call *) t_alloc( fd, T_CALL, T_ADDR );

       if ( callptr == NULL  )
       { t_error( argv[0] );  exit(3); }

       /* Start listening */
       if ( t_listen( fd, callptr ) < 0 )
             { t_error( argv[0] );  exit(4); }

       /* Call has come in or wouldn't be here. Open "talking" endpoint */
       if ((newfd=t_open( DEV_TCP, O_RDWR, (struct t_info *)0)) < 0)
       { t_error( argv[0] );  exit(5); }

       /* Bind to that endpoint*/
       if (t_bind(newfd,(struct t_bind *)0,(struct t_bind *)0) < 0 )
            { t_error( argv[0] );  exit(6); }

       /* Accept the call */
       if ( t_accept( fd, newfd, callptr )  < 0 )
       {
            if ( t_errno == TLOOK )
            {
           if ( t_rcvdis( fd, (struct t_discon *) 0 ) < 0 )
                { t_error ( argv[0] );  exit(7); }
```

```
            }
        t_error( argv[0] );  exit(8);
      }

if ((event = t_rcv(newfd, buffer, 10, &flags)) <0)
  {t_error("t_rcv in server"); exit(8);}
c = atoi(buffer);
for (i = 0; i < c * 1024;  i++)
   DATA[i] = 'Z';
DATA[i] = 0;
i = t_snd( newfd, DATA, strlen(DATA), 0 ) ;
sleep(9); /* Sleep to hold connection open until all data transferred */
t_close( fd );
t_close( newfd );}
}

#################################################################

/*********************************************************
 * FILE - tliclnt.c - client for TLI performance testing *
 *********************************************************/

#include "header.h" /*  TLI definitions    */
#define MAX 140000

main( argc, argv )  /*  arg count, arg vector     */

char *argv[];
{
FILE *g;
char buffer[MAX];
char input[5];
struct sockaddr_in server;  /*  Internet address     */
struct t_call *callptr;     /* TLI call request */
char   *t_alloc();     /* TLI allocate function */
int  c,b, fd, i, event, flags;     /* transport
                                          endpoint    */
long start, finish;
struct timeval before, after;
struct timezone tz;

gettimeofday(&before, &tz); /* Get start time */

if ((fd = t_open( DEV_TCP, O_RDWR, (struct t_info *)0 )) < 0)
     {  t_error( argv[0] );  exit(1); }

if ( t_bind(fd, (struct t_bind *)0, (struct t_bind *)0 ) < 0)
     { t_error( argv[0] );  exit(2); }

memset( &server, 0, sizeof(server));  /* POSIX function */
server.sin_family = AF_INET;          /* Internet domain */
server.sin_port = htons(TCP_PORT); /* "advertised" port */
server.sin_addr.s_addr = inet_addr( argv[1] );
callptr = (struct t_call *) t_alloc( fd, T_CALL, T_ADDR );
if ( callptr == NULL ) { t_error( argv[0] );  exit(3); }
callptr->addr.maxlen = callptr->addr.len = sizeof(server);
callptr->addr.buf    = (char *) &server;
callptr->opt.len     = 0; /* no special options         */
callptr->udata.len   = 0; /* no user data w/connect req. */

if ( t_connect( fd, callptr, (struct t_call *) 0 ) < 0 )
     { t_error( argv[0] );  exit(4); }
if ((event = t_snd(fd, argv[2], strlen(argv[2]), 0)) < 0)
   {t_error(argv[0]); exit(8);}
```

```
b = 0; /* Variable to hold to total */
i = atoi(argv[2]); /* Convert argument into integer */

/* Receive data.  As soon as all received, break connection */

while (b != i *  1024 && event != -1)
{
event = t_rcv(fd, buffer, MAX, &flags);
b = b + event;
}
t_close( fd );

gettimeofday(&after, &tz); /* Get ending time */
start = (before.tv_sec * 1000000 ) + before.tv_usec;
finish = (after.tv_sec * 1000000) + after.tv_usec;
g = fopen("timefile","w");
if (g == NULL)
      {     printf("Cannot open the timefile  \n");
            exit(1);
      }
fprintf(g, "%ld \n",finish - start); /* Write to file */
fclose(g);

}
```

##

See APPENDIX F for a listing of the header.h file

APPENDIX D

TESTING AVAILABILITY OF TRANSPORT SERVICES - SOURCE CODE

```
/*************************************************************
 * FILE - tcptest.c                                          *
 * Program tests which protocols are available on a system *
/*************************************************************/

#include <fcntl.h>
#include <stdio.h>
#include <tiuser.h>
/* #include <xti.h> */

struct transpt
  {  char *name;
     struct t_info tin;
     int fd;
  }

tp [] =

 {  { "/dev/tcp",     {0, 0, 0, 0, 0, 0, 0 }, 0 },
    { "/dev/udp",     {0, 0, 0, 0, 0, 0, 0 }, 0 },
    { "TOSITP0",      {0, 0, 0, 0, 0, 0, 0 }, 0 },
    { "TOSITP2",      {0, 0, 0, 0, 0, 0, 0 }, 0 },
    { "TOSITP4",      {0, 0, 0, 0, 0, 0, 0 }, 0 },
    {  NULL ,         {0, 0, 0, 0, 0, 0, 0 }, 0 }
 };

main()
  {  int k = 0;

    printf("\n%14s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\n",
           "name", "addr", "options", "tsdu",
           "etsdu", "connect", "discon", "servtype");

    while(  tp[k].name != NULL)
     {if ((tp[k].fd=t_open(tp[k].name,O_RDWR,&tp[k].tin)) < 0)
        t_error( tp[k].name);
      else
            {printf("%14s", tp[k].name);
             printf("\t%d",tp[k].tin.addr);
             printf("\t%d",tp[k].tin.options);
             printf("\t%d",tp[k].tin.tsdu);
             printf("\t%d",tp[k].tin.etsdu);
             printf("\t%d",tp[k].tin.connect);
             printf("\t%d",tp[k].tin.discon);
             printf("\t%d",tp[k].tin.servtype);

/* Interprets integer service codes and prints a mnemonic description of the
service type. */

           switch( tp[k].tin.servtype )
           {
            case T_COTS:    printf(" COTS");  break;
            case T_CLTS:    printf(" CLTS");  break;
            case T_COTS_ORD: printf(" COTS_ORD"); break;
            default: printf("unknown!")    ; break;
            }
            printf( "\n");
          }
       k++;
     }
    printf( "\n");
}
```

APPENDIX E

ONC RPC FILE TRANSFER - SOURCE CODE

```
/*********************************************************
 * FILE - trans.x  - ON both IBM and Sun                 *
 *                                                       *
 * User generated-----                                   *
 * Protocol specification file for RPC transfer program *
 *********************************************************/

const MAXNAMELEN = 255;

typedef string name<MAXNAMELEN>;

struct file {
int totallength; /* Total length of one transfer */
int more;        /* Flag for whether more data */
char data<>;     /* Array for data */
};

program TRANSPROG {
     version TRANSVERS {
          file READFILE(name) = 1;
     } = 1;
}= 0x20000009;

#################################################################

/*********************************************************
 * FILE - trans_svc_proc.c  - ON Both IBM and Sun        *
 *                                                       *
 * User-written server RPC program.  This program opens the *
 * source file, reads the data into the data array, and  *
 * writes the data to the target file.                   *
 *********************************************************/

#include <rpc/rpc.h> /* RPC-specific information */
#include "trans.h"   /* RPCGEN-generated include file */
#include <stdio.h>

{

int i, c;/* Useful variable */
file *
readfile_1(msg)
          name *msg;

{
      FILE *f;
      static file openfile; /* Structure to hold return info */
      char hold[140000]; /* Array for data */
      int firsttime;
      while(1)
      {
      if(firsttime == 0) f = fopen(*msg, "r");
            firsttime = 1;
            if (f == NULL)
            {printf("Error in opening the file %s /n",*msg);
                  exit(2);
                  }
            i = 0;
      while ((c != EOF) && (i < 140000))
            {
                  c = getc(f);
                  hold[i] = c;
                  if (c!= EOF) i++;
            }
            hold[i] = 0;
```

```
            if (c == EOF) openfile.more = 0;
        else openfile.more = 1;
              openfile.data.data_val = hold;
              openfile.data.data_len = strlen(hold);
              if (c == EOF) fclose(f);
        return(&openfile);

        }
      firsttime = 0;
}

##################################################################

/*************************************************************
 * FILE  - trans.c  - Both IBM and Sun                       *
 *                                                           *
 * User generated ----                                       *
 * Client program for the RPC program which transfers        *
 * files, both binary and ASCII.  Parameters are:            *
 * ClntPgmName SrvrNameorIPAddress SourceFile TargetFile     *
 *************************************************************/

#include <stdio.h>
#include <rpc/rpc.h>
#include "trans.h"

main(argc, argv)

int argc;
char *argv[];

{
        FILE *g;            /* Pointer to file for writing  */
        CLIENT *cl;  /* File pointer for client      */
        char *server,       /* File pointer for server      */
              *fn,   /* File pointer for source file */
              *newfile;     /* File pointer for target file */
        file *result;       /* Pointer for returned result  */
        int i, b, c, a;     /* Variables for various purposes */

        char hold[140000]; /*Temporary array */

        server = argv[1];  /* Assign argv[1] as server */
        fn = argv[2];             /* Assign argv[2] as source file */
        newfile = argv[3];        /* Assign argv[3] as target file */

        /* create the connection to the server */
        cl = clnt_create(server, TRANSPROG, TRANSVERS, "tcp");

        /* report an error */
        if (cl == NULL)
              { clnt_pcreateerror(server);
             exit(1);
            }

        /* Open the file for write */
        g = fopen(newfile, "w");

      if (g == NULL)
        { printf("Cannot open the file %s \n",newfile);
                exit(1);
              }

        server = argv[1];
        a = 1;
```

```
        while (a != 0)
        {
                result = readfile_1(&fn,cl); /*Remote read */
                if (result == NULL)
                        clnt_perror(cl,server);
                b = result -> data.data_len;
                        for (i=0; i < b ; i++)
                putc( result -> data.data_val[i], g);
                a = result -> more;
        }
        fclose(g);
}

##############################################################

/*********************************************************
 * FILE - trans.h  - From Sun                            *
 * Please do not edit this file.                         *
 * It was generated using rpcgen.                        *
 *********************************************************/

#include <rpc/types.h>

#define MAXNAMELEN 255

typedef char *name;
bool_t xdr_name();

struct file {
        int totallength;
        int more;
        struct {
                u_int data_len;
                char *data_val;
        } data;
};
typedef struct file file;
bool_t xdr_file();

#define TRANSPROG ((u_long)0x20000009)
#define TRANSVERS ((u_long)1)
#define READFILE ((u_long)1)
extern file *readfile_1();

##############################################################

/*********************************************************
 * FILE - trans_svc.c - From Sun                         *
 * Please do not edit this file.                         *
 * It was generated using rpcgen.                        *
 *********************************************************/

#include <stdio.h>
#include <rpc/rpc.h>
#include "trans.h"

static void transprog_1();

main()
{
        register SVCXPRT *transp;

        (void) pmap_unset(TRANSPROG, TRANSVERS);

        transp = svcudp_create(RPC_ANYSOCK);
```

```
        if (transp == NULL) {
                fprintf(stderr, "cannot create udp service.");
                exit(1);
        }
        if    (!svc_register(transp,    TRANSPROG,    TRANSVERS,    transprog_1,
IPPROTO_UDP)) {
                fprintf(stderr,  "unable  to  register  (TRANSPROG,  TRANSVERS,
udp).");
                exit(1);
        }

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL) {
                fprintf(stderr, "cannot create tcp service.");
                exit(1);
        }
        if    (!svc_register(transp,    TRANSPROG,    TRANSVERS,    transprog_1,
IPPROTO_TCP)) {
                fprintf(stderr,  "unable  to  register  (TRANSPROG,  TRANSVERS,
tcp).");
                exit(1);
        }

        svc_run();
        fprintf(stderr, "svc_run returned");
        exit(1);
        /* NOT REACHED */
}

static void
transprog_1(rqstp, transp)
        struct svc_req *rqstp;
        register SVCXPRT *transp;
{
        union {
                name readfile_1_arg;
        } argument;
        char *result;
        bool_t (*xdr_argument)(), (*xdr_result)();
        char *(*local)();

        switch (rqstp->rq_proc) {
        case NULLPROC:
                (void) svc_sendreply(transp, xdr_void, (char *)NULL);
                return;

        case READFILE:
                xdr_argument = xdr_name;
                xdr_result = xdr_file;
                local = (char *(*)()) readfile_1;
                break;

        default:
                svcerr_noproc(transp);
                return;
        }
        bzero((char *)&argument, sizeof(argument));
        if (!svc_getargs(transp, xdr_argument, &argument)) {
                svcerr_decode(transp);
                return;
        }
        result = (*local)(&argument, rqstp);
        if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
                svcerr_systemerr(transp);
        }
```

```
        if (!svc_freeargs(transp, xdr_argument, &argument)) {
                fprintf(stderr, "unable to free arguments");
                exit(1);
        }
        return;
}

##############################################################
/*****************************************************
 * FILE - trans_clnt.t - From Sun                    *
 * Please do not edit this file.                     *
 * It was generated using rpcgen.                    *
 *****************************************************/

#include <rpc/rpc.h>
#include "trans.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

file *
readfile_1(argp, clnt)
        name *argp;
        CLIENT *clnt;
{
        static file res;

        bzero((char *)&res, sizeof(res));
        if (clnt_call(clnt, READFILE, xdr_name, argp, xdr_file, &res, TIMEOUT)
!= RPC_SUCCESS) {
                return (NULL);
        }
        return (&res);
}

##############################################################

/*****************************************************
 * FILE - trans_xdr.c - From Sun                     *
 * Please do not edit this file.                     *
 * It was generated using rpcgen.                    *
 *****************************************************/

#include <rpc/rpc.h>
#include "trans.h"

bool_t
xdr_name(xdrs, objp)
        XDR *xdrs;
        name *objp;
{
        if (!xdr_string(xdrs, objp, MAXNAMELEN)) {
                return (FALSE);
        }
        return (TRUE);
}

bool_t
xdr_file(xdrs, objp)
        XDR *xdrs;
        file *objp;
{
        if (!xdr_int(xdrs, &objp->totallength)) {
                return (FALSE);
        }
```

```
        if (!xdr_int(xdrs, &objp->more)) {
                return (FALSE);
        }
        if    (!xdr_array(xdrs,    (char    **)&objp->data.data_val,    (u_int
              *)&objp->data.data_len, ~0, sizeof(char), xdr_char)) {
                return (FALSE);
        }
        return (TRUE);
}
```

APPENDIX F

TLI FILE TRANSFER - SOURCE CODE

```
/*********************************************************************
 *  FILE - server.c                                                  *
 *  This program uses the streams interface (TLI) to transfer files *
 *********************************************************************/

extern int t_errno ;  /*  TLI library errorcode          */
#include "header.h"   /*  TLI definitions    */
#define MAX 14000
main( argc, argv )
char *argv[];
{
FILE *f; /* Pointer for the source file */
int timeinloop, /* Registers time in write loop */
int    fd, /* "Listening" endpoint */
       newfd,
      b,i,
      flags,
      event,
      c,
      more;
struct sockaddr_in client, server; /* Internet addresses    */

struct t_bind    req;       /* TLI struct for bind req    */
struct t_call    *callptr;  /* TLI struct for call req    */
char buffer[100]; /* To hold source file name */
int   DATA[MAX];     /* data buffer area      */

while(1)

/* Open the endpoint */
{
if ((fd = t_open(DEV_TCP, O_RDWR, (struct t_info *)0 )) < 0 )
     {  t_error( argv[0] );  exit(1); }

memset( &server, 0, sizeof(server)); /* Determine server size */
server.sin_family = AF_INET;          /* Internet domain  */
server.sin_port   = htons(TCP_PORT);  /* Advertised port  */
server.sin_addr.s_addr = htonl(INADDR_ANY); /* any board   */

 /* req.addr.maxlen = maximum bytes buf can hold */
/* req.addr.buf = buffer containing address */
/* req.addr.len = size of the address */
/* req.qlen = maximum connections to support */
req.addr.maxlen  =  req.addr.len  =  sizeof( server );
req.addr.buf     = (char *) &server ;  req.qlen  =  5;

/* Bind to the "listening endpoint */
  if ( t_bind ( fd, &req, (struct t_bind *) 0 ) < 0 )
     { t_error( argv[0] );  exit(2); }

callptr = (struct t_call *) t_alloc( fd, T_CALL, T_ADDR );
if ( callptr == NULL  )
     { t_error( argv[0] );  exit(3); }

/* Start listening */
if ( t_listen( fd, callptr ) < 0 )
     { t_error( argv[0] );  exit(4); }

/* Call has come in or wouldn't be here. Open "talking" endpoint */
if ((newfd=t_open( DEV_TCP, O_RDWR, (struct t_info *)0)) < 0)
     { t_error( argv[0] );  exit(5); }

/* Bind to that endpoint*/
if (t_bind(newfd,(struct t_bind *)0,(struct t_bind *)0) < 0 )
     { t_error( argv[0] );  exit(6); }
```

```
/* Accept the call */
if ( t_accept( fd, newfd, callptr )  < 0 )
     {
       if ( t_errno == TLOOK )
          {
            if ( t_rcvdis( fd, (struct t_discon *) 0 ) < 0 )
               { t_error ( argv[0] );  exit(7); }
          }
       t_error( argv[0] );  exit(8);
     }
/* Receive the source file name from client */
if ((event = t_rcv(newfd, buffer, strlen(buffer), &flags)) <0)
  {t_error("t_rcv in server"); exit(8);}

/* Open the read file */
f = fopen(buffer, "r");

/* Check for error */
if (f == NULL) { printf("Error in opening file %s \n",buffer); exit(9);}
c = getc(f);
for (i = 0; i < MAX; i++)
  DATA[i] = 0;
while ((c != EOF))
   {
     i = 0;
     while ((i < 14000) && (c != EOF))
    {
       DATA[i] = c;
       c = getc(f);
       if (c == EOF) {
             i++;
             DATA[i] = c;
              }
       i++;}
DATA[i] = 0;

i = t_snd( newfd, DATA, strlen(DATA), 0 ) ;
sleep(1);
 }
/* Close all files */
fclose(f);
t_close( fd );
t_close( newfd );}
}

##############################################################################

/*****************************************************************************
 * FILE -  client.c                                                          *
 * Client program using TLI to transfer a file                               *
 *****************************************************************************/

#include "header.h" /*  TLI definitions    */
#define MAX 140000

format() /* procedure to display proper format */

{  printf("Command format is: \n\n");
   printf("CLIENT IpAddress SourceFile  TargetFile \n\n");
   printf("i.e., CLIENT 193.193.193.193 server1.c test1 \n");
}

main( argc, argv )
```

```
char *argv[];
{
FILE *g; /* Target file */
int totalcount, /* Count of characters in file */
    timeinloop; /* Time in write loop */
int buffer[MAX];  /* Receive buffer */
struct sockaddr_in server;  /*  Internet address     */
struct t_call *callptr;     /* TLI call request */
char   *t_alloc();     /* TLI allocate function */
int  c,b,i,  /* Variables for generic purposes */
      fd,          /* File endpoint */
      event,            /* Holds results of calls */
      flags;     /* If more data, used in connectionless */

if (argc != 4)
      {  format();
         exit(6);
      }
if ((fd = t_open(DEV_TCP,O_RDWR,(struct t_info *)0)) < 0)
      {  t_error( argv[0] );  exit(1); }
if (t_bind(fd,(struct t_bind *)0,(struct t_bind *)0) < 0)
      {  t_error( argv[0] );  exit(2); }

memset( &server, 0, sizeof(server));  /* Server size */
server.sin_family = AF_INET; /* Internet domain  */
server.sin_port   = htons(TCP_PORT);/* Advertised port */

/* Assign the IP address for the server */
server.sin_addr.s_addr = inet_addr( argv[1] );

callptr = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR );

if (callptr == NULL )
       { t_error( argv[0] );  exit(3); }
callptr->addr.maxlen = callptr->addr.len = sizeof(server);

callptr->addr.buf = (char *) &server;

callptr->opt.len = 0;    /* no special options          */
callptr->udata.len = 0; /* no user data w/connect req.*/

if (t_connect(fd,callptr,(struct t_call *) 0) < 0)
      { t_error( argv[0] );  exit(4); }

if ((event = t_snd( fd, argv[2], 15, 0)) < 0)
      {t_error(argv[0]); exit(8);}

g = fopen( argv[3],"w");
if (g == NULL)
      { printf(" Error in opening file %s \n", argv[3]);
        exit(9); }
b = 0;
/* Receive the data */
while ((event = t_rcv(fd, buffer, MAX, &flags)) > 0)
{
i = 0;
      while ((buffer[i] != EOF)  && (i < event  ))
       {
            putc(buffer[i],g);
            i++;
         b++;
       }
}
fclose(g);
t_close( fd );
```

```
}

##############################################################

/*************************************************************
 * FILE -  header.h                                          *
 * Includes standard include files required                  *
 *************************************************************/

#include <stdio.h>
#include <fcntl.h>   /* O_RDWR,  O_NDELAY    */
#include <tiuser.h>  /* standard TLI definitions     */
#include <sys/time.h>
#include <errno.h>
#include <time.h>
#include <sys/types.h>
#include <sys/socket.h>  /* Socket  definitions */
#include <netinet/in.h>  /* Internet definitions */
#define DEV_TCP  "/dev/tcp"  /* Device file */
#define TCP_PORT 5255   /* port number for this service */
```

APPENDIX G

DCE RPC FILE TRANSFER - SOURCE CODE

```
/*************************************************************
 * FILE - transfer.idl                                       *
 *                                                           *
 * User-generated --------                                   *
 * Protocol specification file from which are generated:     *
 * 1. C language header file containing definitions          *
 *    needed  by stubs and application code.                 *
 * 2. Client stub; linked with client part of application.   *
 * 3. Server stub; linked with server portion.               *
 *                                                           *
 * Code is run through IDL compiler to generate other code   *
 *                                                           *
 * A Unique uuid which is generated by the uuidgen utility   *
*************************************************************/

[
uuid(005217be-88ac-1d5d-aa3d-08005a4d0145),
version(1.0)
]

/* The chosen interface name (transfer) */

interface transfer
{

typedef [string] char name[255];
typedef [string] char data1[140000];

/* Procedure declaration. void does not return a value; An */
/* out directional attribute must be a pointer or an array */
/* so that the parameter can be passed to the client stub */
/* by reference. Input parameters may be passed by value. */
/*********************************************************/

void readfile(
[in] name *infile, /* The file name of the remote file */
[out] data1 somedata /* The array of file data */
  );
}

#############################################################

/*********************************************************
 * FILE - server.c                                       *
 *                                                       *
 * Adopted from server.c found in [63]                   *
 *                                                       *
 * This program readies the server                       *
 *********************************************************/

#include <stdio.h>
#include "transfer.h" /* header created by IDL compiler */
#include "../arithmetic/check_status.h" /* header with the CHECK_STATUS macro
*/

main ()
{
unsigned32 status; /* error status (nbase.h) */
rpc_binding_vector_t *binding_vector; /*set of binding handles(rpcbase.h)*/
unsigned_char_t *entry_name; /*entry name for name service (lbase.h)*/
char *getenv();

rpc_server_register_if(  /* register interface with RPC runtime */
transfer_v1_0_s_ifspec,   /* interface specification */
      NULL,
```

```
      NULL,
      &status              /* error status */
   );
CHECK_STATUS(status, "Can't register interface\n", ABORT);

rpc_server_use_all_protseqs( /* create binding information */
rpc_c_protseq_max_reqs_default,   /* queue size for calls (rpcbase.h) */
      &status
   );
CHECK_STATUS(status, "Can't create binding information\n", ABORT);

rpc_server_inq_bindings( /* obtain this server's binding information */
     &binding_vector,
     &status
   );
CHECK_STATUS(status, "Can't get binding information\n", ABORT);

entry_name = "/.:/pricepc/transfer_machine";
rpc_ns_binding_export( /* export entry to name service database  */

rpc_c_ns_syntax_default, /* syntax of the entry name  (rpcbase.h) */
entry_name,  /* entry name for name service           */
transfer_v1_0_s_ifspec,    /* interface specification */
      binding_vector,  /* set of server binding handles */
      NULL,
      &status
   );
CHECK_STATUS(status, "Can't export to name service database\n", ABORT);

rpc_ep_register(           /* register endpoints in local endpoint map */
  transfer_v1_0_s_ifspec,   /* interface specification */
      binding_vector,  /* the set of server binding handles */
      NULL,
      NULL,
      &status
   );
CHECK_STATUS(status, "Can't add address to the endpoint map\n", ABORT);

rpc_binding_vector_free( /* free set of server binding handles */
      &binding_vector,
      &status
   );
CHECK_STATUS(status, "Can't free binding handles and vector\n", ABORT);

puts("Listening for remote procedure calls...");
rpc_server_listen( /* listen for remote calls  */
      rpc_c_listen_max_calls_default, /*concurrent calls to
                                       server (rpcbase.h)*/
      &status
   );
   CHECK_STATUS(status, "rpc listen failed\n", ABORT);
}

#############################################################

/*************************************************************
 * FILE - procedure.c                                        *
 *                                                           *
 * User-generated server code for DCE file transfer          *
 * application.                                              *
 * Upon connection by a client, it receives the file         *
 * name on the remote system for transfer, and sends         *
 * blocks of the file until the entire file is transferred */
 *************************************************************/
```

```
#include "transfer.h" /* IDL-generated header file */
#include <stdio.h>
#include <string.h>

void readfile(infile,somedata)
name *infile;
data1 somedata;

{
      int i, c, b; /* various miscellaneous-use variables */
      FILE *f;     /* pointer for file to be transferred */
      int firsttime; /* indicator for whether to open file */

      if(firsttime == 0) f = fopen(*infile, "r");
      firsttime = 1;
      if (f == NULL)
      {printf("Error in opening the file %s \n",*infile);
          exit(2);
         }
      i = 0;
     while ((c != EOF) && (i < 140000)) /* limit set by
                                           machine memory */
      {
      c = getc(f);
            somedata[i] = c;
            if (c != EOF) i++;
     }
      somedata[i] = 0;
      if (c == EOF)
      {
            fclose(f);
            firsttime = 0; /* Allow another file transfer */
      }

}

##############################################################

/*************************************************************
 * FILE - client.c                                           *
 *                                                           *
 * User-generated ---                                        *
 * Client program for DCE file transfer application.         *
 * Receives as user input the names of the remote file for   *
 * transfer and the local file to which it will be transfer- *
 * red. Requests are made of the server until the entire     *
 * file is transferred                                       *
 *************************************************************/

#include <stdio.h>
#include "transfer.h" /* IDL-generated header file */

main(argc, argv)

int argc;
char *argv[];

{
      FILE *g; /* Pointer to the local file transfer */
      char *fn,  /* the remote file name */
           *newfile;  /* The new local file name */
      data1 data; /* The array to hold transferred data */
      int i, /* Counter variable */
            a;  /* Flag to signal no more data */
```

```
        fn = argv[1];
        newfile = argv[2];
        g = fopen(newfile, "w");
        if (g == NULL)
        { printf("Cannot open the file %s \n",newfile);
                exit(1);
        }
        a = 1;
        while (a != 0)
        {
              readfile(fn,&data); /* Make the remote call */
              a = strlen(&data); /* See how much data was sent */
              if (a != 0)
                    for (i=0; i< a ; i++)
                          fputc(data[i],g); /* Write to file */

        }
fclose(g);
}

##############################################################

/*****************************************************
 * File - transfer.h                                 *
 * Generated by IDL compiler version DCE 1.0.0-1     *
 *****************************************************/


#ifndef transfer_v1_0_included
#define transfer_v1_0_included
#include <dce/idlbase.h>
#include <dce/rpc.h>

#ifdef __cplusplus
    extern "C" {
#endif

#include <dce/nbase.h>
typedef idl_char name[255];
typedef idl_char data1[140000];
extern void readfile(
#ifdef IDL_PROTOTYPES
    /* [in] */ name (*infile),
    /* [out] */ data1 somedata
#endif
);
typedef struct transfer_v1_0_epv_t {
void (*readfile)(
#ifdef IDL_PROTOTYPES
    /* [in] */ name (*infile),
    /* [out] */ data1 somedata
#endif
);
} transfer_v1_0_epv_t;
extern rpc_if_handle_t transfer_v1_0_c_ifspec;
extern rpc_if_handle_t transfer_v1_0_s_ifspec;

#ifdef __cplusplus
    }
#endif

#endif
```

VITA

Patricia Chapin Price was born in Hendersonville, North Carolina on November 1, 1951. She graduated from Charles D. Owen High School in Swannanoa, North Carolina in June, 1970. The following September she entered The University of North Carolina at Greensboro and in May, 1974 received the degree of Bachelor of Arts in Biology. She entered The University of North Carolina at Asheville in January, 1982 and in May, 1984 received a Bachelor of Science degree in Computer Science. In January, 1989 she entered the University of Tennessee at Knoxville and in May, 1994 received a Master of Science degree in Computer Science.

She is presently employed by Martin Marietta Energy Systems, Incorporated and is physically sited at the Oak Ridge National Laboratory in Oak Ridge, Tennessee. She is a member of the Advanced Technology Integration and Development Department where she assists in the evaluation of emerging networking technologies and their potential use in the fulfillment of the corporate mission of her employer.