A high-level framework for network-based resource sharing*

by JAMES E. WHITE
Stanford Research Institute
Menlo Park, California

ABSTRACT

This paper proposes a high-level, application-independent framework for the construction of distributed systems within a resource sharing computer network. The framework generalizes design techniques in use within the ARPA Computer Network. It eliminates the need for application-specific communication protocols and support software, thus easing the task of the applications programmer and so encouraging the sharing of resources. The framework consists of a network-wide protocol for invoking arbitrary named functions in a remote process, and machine-dependent system software that interfaces one applications program to another via the protocol. The protocol provides mechanisms for supplying arguments to remote functions and for retrieving their results; it also defines a small number of standard data types from which all arguments and results must be modeled. The paper further proposes that remote functions be thought of as remotely callable subroutines or procedures. This model would enable the framework to more gracefully extend the local programming environment to embrace modules on other machines.

THE GOAL, RESOURCE SHARING

The principal goal of all resource-sharing computer networks, including the now international ARPA Network (the ARPANET), is to usefully interconnect geographically distributed hardware, software, and human resources.[1] Achieving this goal requires the design and implementation of various levels of support software within each constituent computer, and the specification of network-wide "protocols" (that is, conventions regarding the format and the relative timing of network messages) governing their interaction. This paper outlines an alternative to the approach that ARPANET system builders have been taking since work in this area began in 1970, and suggests a strategy for modeling distributed systems within any large computer network.

* The work reported here was supported by the Advanced Research Projects Agency of the Department of Defense, and by the Rome Air Development Center of the Air Force.

The first section of this paper describes the prevailing ARPANET protocol strategy, which involves specifying a family of application-dependent protocols with a network-wide inter-process communication facility as their common foundation. In the second section, the application-independent command/response discipline that characterizes this protocol family is identified and its isolation as a separate protocol proposed. Such isolation would reduce the work of the applications programmer by allowing the software that implements the protocol to be factored out of each applications program and supplied as a single, installation-maintained module. The final section of this paper proposes an extensible model for this class of network interaction that in itself would even further encourage the use of network resources.

THE CURRENT SOFTWARE APPROACH TO RESOURCE SHARING

Function-oriented protocols

The current ARPANET software approach to facilitating resource sharing has been detailed elsewhere in the literature.[2,3,4] Briefly, it involves defining a Host-Host Protocol by which the operating systems of the various "host" computers cooperate to support a network-wide inter-process communication (IPC) facility, and then various function-oriented protocols by which processes deliver and receive specific services via IPC. Each function-oriented protocol regulates the dialog between a resident "server process" providing the service, and a "user process" seeking the service on behalf of a user (the terms "user" and "user process" will be used consistently throughout this paper to distinguish the human user from the computer process acting on his behalf).

The current Host-Host Protocol has been in service since 1970. Since its initial design and implementation, a variety of deficiencies have been recognized and several alternative protocols suggested.[5,6] Although improvements at this level would surely have a positive effect upon Network resource sharing, the present paper simply assumes the existence of some form of

IPC and focuses attention upon higher level protocol design issues.

Each of the function-oriented protocols mentioned in this paper constitutes the official ARPANET protocol for its respective application domain and is therefore implemented at nearly all of the 75 host installations that now comprise the Network. It is primarily upon this widely implemented protocol family (and the philosophy it represents) that the present paper focuses. Needless to say, other important resource-sharing tools have also been constructed within the ARPANET. The Resource-Sharing Executive (RSEXEC), designed and implemented by Bolt, Beranek and Newman, Inc.,[7] provides an excellent example of such work.

Experience with and limitations of hands-on resource sharing

The oldest and still by far the most heavily used function-oriented protocol is the Telecommunications Network protocol (TELNET),[8] which effectively attaches a terminal on one computer to an interactive time-sharing system on another, and allows a user to interact with the remote system via the terminal as if he were one of its local users.

As depicted in Figure 1, TELNET specifies the means by which a user process monitoring the user's terminal is interconnected, via an IPC communication channel, with a server process with access to the target time-sharing system. TELNET also legislates a standard character set in which the user's commands and the system's responses are to be represented in transmission between machines. The syntax and semantics of these interchanges, however, vary from one system to another and are unregulated by the protocol; the user and server processes simply shuttle characters between the human user and the target system.

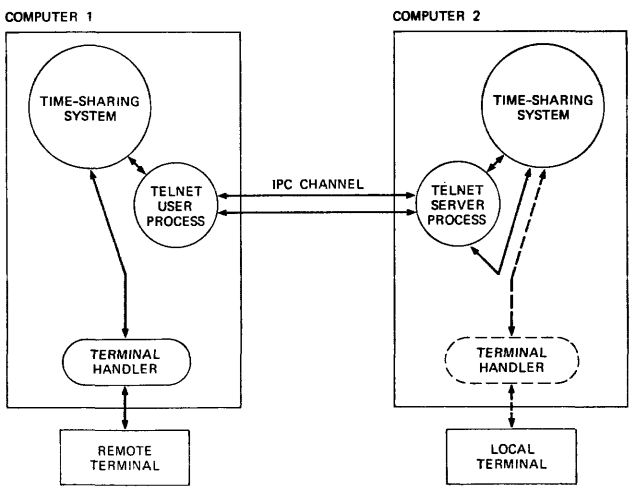

Figure 1—Interfacing a remote terminal to a local time-sharing system via the TELNET Protocol

Although the hands-on use of remote resources that TELNET makes possible is a natural and highly visible form of resource sharing, several limitations severely reduce its long-term utility:

(1) It forces upon the user all of the trappings of the resource's own system.

To exploit a remote resource, the user must leave the familiar working environment provided by his local system and enter an alien one with its own peculiar system structure (login, logout, and subsystem entry and exit procedures) and command language discipline (command recognition and completion conventions, editing characters, and so on). Hands-on resource sharing thus fails to provide the user with the kind of organized and consistent workshop he requires to work effectively.[9]

(2) It provides no basis for bootstrapping new composite resources from existing ones.

Because the network access discipline imposed by each resource is a human-engineered command language, rather than a machine-oriented communication protocol, it is virtually impossible for one resource to programatically draw upon the services of others. Doing so would require that the program deal successfully with complicated echoing and feedback characteristics; unstructured, even unsolicited system responses; and so forth. Hands-on resource sharing thus does nothing to provide an environment in which existing resources can be used as building blocks to construct new, more powerful ones.

These inherent limitations of hands-on resource sharing are removed by a protocol that simplifies and standardizes the dialog between user and server processes. Given such a protocol, the various remote resources upon which a user might wish to draw can indeed be made to appear as a single, coherent workshop by interposing between him and them a command language interpreter that transforms his commands into the appropriate protocol utterances.[10,11] The construction of composite resources also becomes feasible, since each resource is accessible by means of a machine-oriented protocol and can thus be readily employed by other processes within the network.

Standardizing the inter-machine dialog in specific application areas

After the TELNET protocol had been designed and widely implemented within the ARPANET, work began on a family of function-oriented protocols designed for use by programs, rather than human users. Each such protocol standardizes the inter-machine dialog in a particular application area. While TELNET dictates

only the manner in which user and server processes are interconnected via the IPC facility, and the character set in which the two processes communicate once connected, each member of this family specifies in addition the syntax and semantics of the commands and responses that comprise their dialog.

Protocols within this family necessarily differ in substance, each specifying its own application-specific command set. The File Transfer Protocol (FTP),[12] for example, specifies commands for manipulating files, and the Remote Job Entry Protocol (RJE)[13] specifies commands for manipulating batch jobs. Protocols throughout the family are, however, similar in form, each successive family member having simply inherited the physical features of its predecessors. Thus FTP and RJE enforce the same conventions for formulating commands and responses.

This common command/response discipline requires that commands and responses have the following respective formats:

command-name <SP> parameter <CRLF>

response-number <SP> text <CRLF>

Each command invoked by the user process is identified by NAME and is allowed a single PARAMETER. Each response generated by the server process contains a three-digit decimal response NUMBER (to be interpreted by the user process) and explanatory TEXT (for presentation, if necessary, to the user). Response numbers are assigned in such a way that, for example, positive and negative acknowledgments can be easily distinguished by the user process.

FTP contains, among others, the following commands (each listed with one of its possible responses) for retrieving, appending to, replacing, and deleting files, respectively, within the server process file system:

Command	Response
RETR <SP> filename <CRLF>	250 <SP> Beginning transfer. <CRLF>
APPE <SP> filename <CRLF>	400 <SP> Not implemented. <CRLF>
STOR <SP> filename <CRLF>	453 <SP> Directory overflow. <CRLF>
DELE <SP> filename <CRLF>	450 <SP> File not found. <CRLF>

The first three commands serve only to initiate the transfer of a file from one machine to another. The transfer itself occurs on a separate IPC channel and is governed by what amounts to a separate protocol.

Since the general command format admits but a single parameter, multiparameter operations must be implemented as sequences of commands. Thus two commands are required to rename a file:

Command	Response
RNFR <SP> oldname <CRLF>	200 <SP> Next parameter. <CRLF>
RNTO <SP> newname <CRLF>	253 <SP> File renamed. <CRLF>

A COMMAND/RESPONSE PROTOCOL, THE BASIS FOR AN ALTERNATIVE APPROACH

The importance of factoring out the command/response discipline

That FTP, RJE, and the other protocols within this family share a common command/response discipline is a fact not formally recognized within the protocol literature, and each new protocol document describes it in detail, as if for the first time. Nowhere are these conventions codified in isolation from the various contexts in which they find use, being viewed as a necessary but relatively unimportant facet of each function-oriented protocol. "This common command/response discipline has thus gone unrecognized as the important, application-independent protocol that it is."

This oversight has had two important negative effects upon the growth of resource sharing within the ARPANET:

(1) It has allowed the command/response discipline to remain crude.

As already noted, operations that require more than a single parameter are consistently implemented as two or more separate commands, each of which requires a response and thus incurs the overhead of a full round-trip network delay. Furthermore, there are no standards for encoding parameter types other than character strings, nor is there provision for returning results in a command response.

(2) It has placed upon the applications programmer the burden of implementing the network "run-time environment (RTE)" that enables him to access remote processes at the desired, functional level.

Before he can address remote processes in terms like the following:

execute function DELE with argument TEXT-FILE on machine X

the applications programmer must first construct (as he invariably does in every program he writes) a module that provides the desired program interface while implementing the agreed upon command/response discipline. This run-time environment contains the code required to properly format outgoing commands, to interface with the IPC facility, and to parse incoming responses. Because the system provides only the IPC facility as a foundation, the

applications programmer is deterred from using remote resources by the amount of specialized knowledge and software that must first be acquired.

If, on the other hand, the command/response discipline were formalized as a separate protocol, its use in subsequent function-oriented protocols could rightly be anticipated by the systems programmer, and a single run-time environment constructed for use throughout an installation (in the worst case, one implementation per programming language per machine might be required). This module could then be placed in a library and, as depicted in Figure 2, link loaded with (or otherwise made available to) each new applications program, thereby greatly simplifying its use of remote resources.

Furthermore, since enhancements to it would pay dividends to every applications program employing its services, the run-time environment would gradually be augmented to provide additional new services to the programmer.

The thesis of the present paper is that one of the keys to facilitating network resource sharing lies in (1) isolating as a separate protocol the command/response discipline common to a large class of applications protocols; (2) making this new, application-independent protocol flexible and efficient; and (3) constructing at each installation a RTE that employs it to give the applications programmer easy and high-level access to remote resources.

Specifications for the command/response protocol

Having argued the value of a command/response protocol (hereafter termed the Protocol) as the foundation for a large class of applications protocols, there remains the task of suggesting the form that the Protocol might take. There are eight requirements. First, it must reproduce the capabilities of the discipline it replaces:

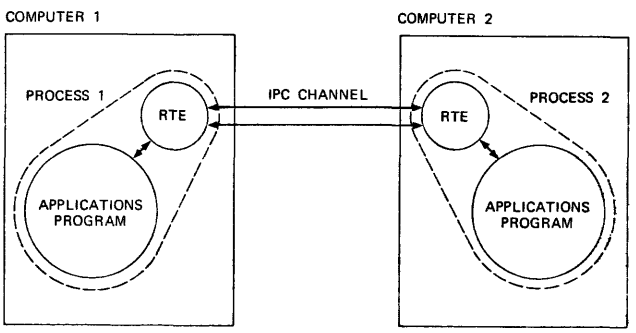

Figure 2—Interfacing distant applications programs via their run-time environments

(1) Permit invocation of arbitrary, named commands (or functions) implemented by the remote process.
(2) Permit command outcomes to be reported in a way that aids both the program invoking the command and the user under whose control it may be executing.

Second, the Protocol should remove the known deficiencies of its predecessor, that is:

(3) Allow an arbitrary number of parameters to be supplied as arguments to a single command.
(4) Provide representations for a variety of parameter types, including but not limited to character strings.
(5) Permit commands to return parameters as results as well as accept them as arguments.

And, finally, the Protocol should provide whatever additional capabilities are required by the more complex distributed systems whose creation the Protocol seeks to encourage. Although others may later be identified, the three capabilities below are recognized now to be important:

(6) Permit the server process to invoke commands in the user process, that is, eliminate entirely the often inappropriate user/server distinction, and allow each process to invoke commands in the other.

In the workshop environment alluded to earlier, for example, the user process is the command language interpreter and the server process is any of the software tools available to the user. While most commands are issued by the interpreter and addressed to the tool, occasionally the tool must invoke commands in the interpreter or in another tool. A graphical text editor, for example, must invoke commands within the interpreter to update the user's display screen after an editing operation.

(7) Permit a process to accept two or more commands for concurrent execution.

The text editor may wish to permit the user to initiate a long formatting operation with one command and yet continue to issue additional, shorter commands before there is a response to the first.

(8) Allow the process issuing a command to suppress the response the command would otherwise elicit.

This feature would permit network traffic to be reduced in those cases in which the process invoking the command deems a response unnecessary. Commands that always succeed but never return results are obvious candidates for this kind of treatment.

A formulation of the protocol that meets these specifications

The eight requirements listed above are met by a protocol in which the following two messages are defined:

message-type=COMMAND [tid] command-name arguments

message-type=RESPONSE tid outcome results

Here and in subsequent protocol descriptions, elements enclosed in square brackets are optional.

The first message invokes the command whose NAME is specified using the ARGUMENTS provided. The second is issued in eventual response to the first and returns the OUTCOME and RESULTS of the completed command. Whenever OUTCOME indicates that a command has failed, the command's RESULTS are required to be an error number and diagnostic message, the former to help the invoking program determine what to do next, the latter for possible presentation to the user. The protocol thus provides a framework for reporting errors, while leaving to the applications program the tasks of assigning error numbers and composing the text of error messages.

There are several elements of the Protocol that are absent from the existing command/response discipline:

(1) RESULTS, in fulfillment of Requirement 5.

(2) A MESSAGE TYPE that distinguishes commands from responses, arising from Requirement 6.

In the existing discipline, this distinction is implicit, since user and server processes receive only responses and commands, respectively.

(3) An optional transaction identifier TID by which a command and its response are associated, arising from Requirements 7 and 8.

The presence of a transaction identifier in a command implies the necessity of a response echoing the identifier; and no two concurrently outstanding commands may bear the same identifier.

Requirements 3 and 4—the ability to transmit an arbitrary number of parameters of various types with each command or response—are most economically and effectively met by defining a small set of primitive "data types" (for example, booleans, integers, character strings) from which concrete parameters can be modeled, and a "transmission format" in which such parameters can be encoded. Appendix A suggests a set of data types suitable for a large class of applications; Appendix B defines some possible transmission formats.

The protocol description given above is, of course, purely symbolic. Appendix C explores one possible encoding of the Protocol in detail.

Summarizing the arguments advanced so far

The author trusts that little of what has been presented thus far will be considered controversial by the reader. The following principal arguments have been made:

(1) The more effective forms of resource sharing depend upon remote resources being usefully accessible to other programs, not just to human users.

(2) Application-dependent protocols providing such access using the current approach leave to the applications programmer the task of constructing the additional layer of software (above the IPC facility provided by the system) required to make remote resources accessible at the functional level, thus discouraging their use.

(3) A single, resource-independent protocol providing flexible and efficient access at the functional level to arbitrary remote resources can be devised.

(4) This protocol would make possible the construction at each installation of an application-independent, network run-time environment making remote resources accessible at the functional level and thus encouraging their use by the applications programmer.

A protocol as simple as that suggested here has great potential for stimulating the sharing of resources within a computer network. First, it would reduce the cost of adapting existing resources for network use by eliminating the need for the design, documentation, and implementation of specialized delivery protocols. Second, it would encourage the use of remote resources by eliminating the need for application-specific interface software. And finally, it would encourage the construction of new resources built expressly for remote access, because of the ease with which they could be offered and used within the network software marketplace.

A HIGH-LEVEL MODEL OF THE NETWORK ENVIRONMENT

The importance of the model imposed by the protocol

The Protocol proposed above imposes upon the applications programmer a particular model of the network environment. In a heterogeneous computer network, nearly every protocol intended for general implementation has this effect, since it idealizes a class of operations that have concrete but slightly different equivalents in each system. Thus the ARPANET's TELNET Protocol alluded to earlier, for example, specifies a Network Virtual Terminal that attempts to provide a best fit to the many real terminals in use around the Network.

As now formulated, the Protocol models a remote resource as an interactive program with a simple, rigidly specified command language. This model follows naturally from the fact that the function-oriented protocols from which the Protocol was extracted were necessitated by the complexity and diversity of user-oriented command languages. The Protocol may thus legitimately be viewed as a vehicle for providing, as an adjunct to the sophisticated command languages already available to users, a family of simple command languages that can readily be employed by programs.

While the command/response model is a natural one, others are possible. A remote resource might also be modeled as a process that services and replies to requests it receives from other computer processes. This request/reply model would emphasize the fact that the Protocol is a vehicle for inter-process communication and that no human user is directly involved.

Substituting the request/reply model for the command/response model requires only cosmetic changes to the Protocol:

message-type=REQUEST [tid] op-code
arguments

message-type=REPLY tid outcome
results

In the formulation above, the terms "REQUEST", "REPLY", and "op-code" have simply been substituted for "COMMAND", "RESPONSE", and "command-name", respectively.

The choice of model need affect neither the content of the Protocol nor the behavior of the processes whose dialog it governs. Use of the word "command" in the command/response model, for example, is not meant to imply that the remote process can be coerced into action. Whatever model is adopted, a process has complete freedom to reject an incoming remote request that it is incapable of or unwilling to fulfill.

But even though it has no substantive effect upon the Protocol, the selection of a model—command/response, request/reply, and so on—is an important task because it determines the way in which both applications and systems programmers perceive the network environment. If the network environment is made to appear foreign to him, the applications programmer may be discouraged from using it. The choice of model also constrains the kind and range of protocol extensions that are likely to occur to the systems programmer; one model may suggest a rich set of useful extensions, another lead nowhere (or worse still, in the wrong direction).

In this final section of the paper, the author suggests a network model (hereafter termed the Model) that he believes will both encourage the use of remote resources by the applications programmer and suggest to the systems programmer a wide variety of useful Protocol extensions. Unlike the substance of the Protocol, however, the Model has already proven quite controversial within the ARPANET community.

Modeling resources as collections of procedures

Ideally, the goal of both the Protocol and its accompanying RTE is to make remote resources as easy to use as local ones. Since local resources usually take the form of resident and/or library subroutines, the possibility of modeling remote commands as "procedures" immediately suggests itself. The Model is further confirmed by the similarity that exists between local procedures and the remote commands to which the Protocol provides access. Both carry out arbitrarily complex, named operations on behalf of the requesting program (the caller); are governed by arguments supplied by the caller; and return to it results that reflect the outcome of the operation. The procedure call model thus acknowledges that, in a network environment, programs must sometimes call subroutines in machines other than their own.

Like the request/reply model already described, the procedure call model requires only cosmetic changes to the Protocol:

message-type=CALL [tid] procedure-name
arguments

message-type=RETURN tid outcome
results

In this third formulation, the terms "CALL", "RETURN", and "procedure-name" have been substituted for "COMMAND", "RESPONSE", and "command-name", respectively. And in this form, the Protocol might aptly be designated a "procedure call protocol (PCP)".

"The procedure call model would elevate the task of creating applications protocols to that of defining procedures and their calling sequences. It would also provide the foundation for a true distributed programming system (DPS) that encourages and facilitates the work of the applications programmer by gracefully extending the local programming environment, via the RTE, to embrace modules on other machines." This integration of local and network programming environments can even be carried as far as modifying compilers to provide minor variants of their normal procedure-calling constructs for addressing remote procedures (for which calls to the appropriate RTE primitives would be dropped out).

Finally, the Model is one that can be naturally extended in a variety of ways (for example, coroutine linkages and signals) to further enhance the distributed programming environment.

Clarifying the procedure call model

Although in many ways it accurately portrays the class of network interactions with which this paper

deals, the Model suggested above may in other respects tend to mislead the applications programmer. The Model must therefore be clarified:

(1) Local procedure calls are cheap; remote procedure calls are not.

Local procedure calls are often effected by means of a single machine instruction and are therefore relatively inexpensive. Remote procedure calls, on the other hand, would be effected by means of a primitive provided by the local RTE and require an exchange of messages via IPC.

Because of this cost differential, the applications programmer must exercise discretion in his use of remote resources, even though the mechanics of their use will have been greatly simplified by the RTE. Like virtual memory, the procedure call model offers great convenience, and therefore power, in exchange for reasonable alertness to the possibilities of abuse.

(2) Conventional programs usually have a single locus of control; distributed programs need not.

Conventional programs are usually implemented as a single process with exactly one locus of control. A procedure call, therefore, traditionally implies a transfer of control from caller to callee. Distributed systems, on the other hand, are implemented as two or more processes, each of which is capable of independent execution. In this new environment, a remote procedure call need not suspend the caller, which is capable of continuing execution in parallel with the called procedure.

The RTE can therefore be expected to provide, for convenience, two modes of remote procedure invocation: a blocking mode that suspends the caller until the procedure returns; and a non-blocking mode that releases the caller as soon as the CALL message has been sent or queued. Most conventional operating systems already provide such a mode choice for I/O operations. For non-blocking calls, the RTE must also, of course, either arrange to asynchronously notify the program when the call is complete, or provide an additional primitive by which the applications program can periodically test for that condition.

Finally, the applications programmer must recognize that by no means all useful forms of network communication are effectively modeled as procedure calls. The lower level IPC facility that remains directly accessible to him must therefore be employed in those applications for which the procedure call model is inappropriate and RTE-provided primitives simply will not do.

SOME EXPECTATIONS

Both the Procedure Call Protocol and its associated Run-Time Environment have great potential for facilitating the work of the network programmer; only a small percentage of that potential has been discussed in the present paper. Upon the foundation provided by PCP can be erected higher level application-independent protocol layers that further enhance the distributed programming environment by providing even more powerful capabilities (see Appendix D).

As the importance of the RTE becomes fully evident, additional tasks will gradually be assigned to it, including perhaps those of:

(1) Converting parameters between the format employed internally by the applications program, and that imposed by the Protocol.
(2) Automatically selecting the most appropriate inter-process transmission format on the basis of the two machines' word sizes.
(3) Automatically substituting for network IPC a more efficient form of communication when both processes reside on the same machine.

The RTE will eventually offer the programmer a wide variety of application-independent, network-programming conveniences, and so, by means of the Protocol, become an increasingly powerful distributed-system-building tool.

ACKNOWLEDGMENTS

Many individuals within both SRI's Augmentation Research Center (ARC) and the larger ARPANET community have contributed their time and ideas to the development of the Protocol and Model described in this paper. The contributions of the following individuals are expressly acknowledged: Dick Watson, Jon Postel, Charles Irby, Ken Victor, Dave Maynard, and Larry Garlick of ARC; and Bob Thomas and Rick Schantz of Bolt, Beranek and Newman, Inc.

ARC has been working toward a high-level framework for network-based distributed systems for a number of years now.[14] The particular Protocol and Model described here result from research begun by ARC in July of 1974. This research included developing the Model; designing and documenting the Protocol required to support it;[15] and designing, documenting, and implementing a prototype run-time environment for a particular machine,[16,17] specifically a PDP-10 running the Tenex operating system developed by Bolt, Beranek and Newman, Inc.[18] Three design iterations were carried out during a 12-month period, and the resulting specification implemented for Tenex. The Tenex RTE provides a superset of the capabilities presented in the body of this paper and Appendices A through C as well as those alluded to in Appendix D.

REFERENCES

1. Kahn, R. E., "Resource-Sharing Computer Communications Networks," *Proceedings of the IEEE*, Vol. 60, No. 11, pp. 1397-1407, November 1972.
2. Crocker, S. D., J. F. Heafner, R. M. Metcalfe and J. B. Postel, "Function-oriented Protocols for the ARPA Computer Network," *AFIPS Proceedings*, Spring Joint Computer Conference, Vol. 40, pp. 271-279, 1972.
3. Carr, C. S., S. D. Crocker and V. G. Cerf, "Host-Host Communication Protocol in the ARPA Network," *AFIPS Proceedings*, Spring Joint Computer Conference, Vol. 36, pp. 589-597, 1970.
4. McKenzie, A. A., *Host/Host Protocol for the ARPA Network*, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, January 1972, SRI-ARC Catalog Item 8246.
5. Walden, D. C., "A System for Interprocess Communication in a Resource Sharing Computer Network," *Communications of the ACM*, Vol. 15, No. 4, pp. 221-230, April 1972.
6. Cerf, V. G. and R. E. Kahn, "A Protocol for Packet Network Intercommunication," *IEEE Transactions on Communications*, Vol. Com-22, No. 5, pp. 637-648, May 1974.
7. Thomas, R. H., "A Resource-Sharing Executive for the ARPANET," *AFIPS Proceedings*, National Computer Conference, Vol. 42, pp. 155-163, 1973.
8. *TELNET Protocol Specification*, Stanford Research Institute, Menlo Park, California, August 1973, SRI-ARC Catalog Item 18639.
9. Engelbart, D. C., R. W. Watson and J. C. Norton, "The Augmented Knowledge Workshop," *AFIPS Proceedings*, National Computer Conference, Vol. 42, pp. 9-21, 1973.
10. Engelbart, D.C. and W. K. English, "A Research Center for Augmenting Human Intellect," *AFIPS Proceedings*, Fall Joint Computer Conference, Vol. 33, pp. 395-410, 1968.
11. Irby, C. H., C. F. Dornbush, K. E. Victor and D. C. Wallace, *A Command Meta Language for NLS*, Final Report, Contract RADC-TR-75-304, SRI Project 1868, Stanford Research Institute, Menlo Park, California, December, 1975.
12. Neigus, N. J., *File Transfer Protocol*, ARPA Network Working Group Request for Comments 542, Bolt Beranek and Newman Inc., Cambridge, Massachusetts, July 1973, SRI-ARC Catalog Item 17759.
13. Bressler, R. D., R. Guida and A. A. McKenzie, *Remote Job Entry Protocol*, ARPA Network Working Group Request for Comments 360, Dynamic Modeling Group, Massachusetts Institute of Technology, Cambridge, Massachusetts, undated, SRI-ARC Catalog Item 12112.
14. Watson, R. W., *Some Thoughts on System Design to Facilitate Resource Sharing*, ARPA Network Working Group Request for Comments 592, Augmentation Research Center, Stanford Research Institute, Menlo Park, California, November 20, 1973, SRI-ARC Catalog Item 20391.
15. White, J. E., *DPS-10 Version 2.5 Implementer's Guide*, Augmentation Research Center, Stanford Research Institute, Menlo Park, California, August 15, 1975, SRI-ARC Catalog Item 26282.
16. White, J. E., *DPS-10 Version 2.5 Programmer's Guide*, Augmentation Research Center, Stanford Research Institute, Menlo Park, California, August 13, 1975, SRI-ARC Catalog Item 26271.
17. White, J. E., *DPS-10 Version 2.5 Source Code*, Augmentation Research Center, Stanford Research Institute, Menlo Park, California, August 13, 1975, SRI-ARC Catalog Item 26267.
18. Bobrow, D. G., J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Communications of the ACM*, Vol. 15, No. 3, pp. 135-143, March 1972.
19. White, J. E., "Elements of a Distributed Programming System," Submitted for publication in the *Journal of Computer Languages*, 1976.

APPENDIX A—SUGGESTED DATA TYPES

The Protocol requires that every parameter or "data object" be represented by one of several primitive data types defined by the Model. The set of data types below is sufficient to conveniently model a large class of data objects, but since the need for additional data types (for example, floating-point numbers) will surely arise, the set must remain open-ended. Throughout the descriptions below, N is confined to the range [0, 2**15-1]:

LIST: A list is an ordered sequence of N data objects called "elements". A LIST may contain other LISTs as elements, and can therefore be employed to construct arbitrarily complex composite data objects.

CHARSTR: A character string is an ordered sequence of N ASCII characters, and conveniently models a variety of textual entities, from short user names to whole paragraphs of text.

BITSTR: A bit string is an ordered sequence of N bits and, therefore, provides a means for representing arbitrary binary data (for example, the contents of a word of memory).

INTEGER: An integer is a fixed-point number in the range [-2**31, 2**31-1], and conveniently models various kinds of numerical data, including time intervals, distances, and so on.

INDEX: An index is an integer in the range [1, 2**15-1]. As its name and value range suggest, an INDEX can be used to address a particular bit or character within a string, or element within a list. INDEXes have other uses as well, including the modeling of handles or identifiers for open files, created processes, and the like. Also, because of their restricted range, INDEXes are more compact in transmission than INTEGERs (see Appendix B).

BOOLEAN: A boolean represents a single bit of information, and has either the value true or false.

EMPTY: An empty is a valueless place holder within a LIST or parameter list.

APPENDIX B—SUGGESTED TRANSMISSION FORMATS

Parameters must be encoded in a standard transmission format before they can be sent from one process to another via the Protocol. An effective strategy is to define several formats and select the most appropriate one at run-time, adding to the Protocol a mechanism for format negotiation. Format negotiation would be another responsibility of the RTE and could thus be made completely invisible to the applications program.

Suggested below are two transmission formats. The first is a 36-bit binary format for use between 36-bit

machines, the second an 8-bit binary, "universal" format for use between dissimilar machines. Data objects are fully typed in each format to enable the RTE to automatically decode and internalize incoming parameters should it be desired to provide this service to the applications program.

PCPB36, For Use Between 36-Bit Machines

Bits 0-13 Unused (zero)
Bits 14-17 Data type

EMPTY	=1	INTEGER	=4 LIST=7
BOOLEAN	=2	BITSTR	=5
INDEX	=3	CHARSTR	=6

Bits 18-20 Unused (zero)
Bits 21-35 Value or length N

EMPTY	unused (zero)
BOOLEAN	14 zero-bits + 1-bit value (TRUE=1/FALSE=0)
INDEX	unsigned value
INTEGER	unused (zero)
BITSTR	unsigned bit count N
CHARSTR	unsigned character count N
LIST	unsigned element count N

Bits 36- Value

EMPTY	unused (nonexistent)
BOOLEAN	unused (nonexistent)
INDEX	unused (nonexistent)
INTEGER	two's complement full-word value
BITSTR	bit string + zero padding to word boundary
CHARSTR	ASCII string + zero padding to word boundary
LIST	element data objects

PCPB8, For Use Between Dissimilar Machines

Byte 0 Data type

EMPTY	=1	INTEGER	=4 LIST=7
BOOLEAN	=2	BITSTR	=5
INDEX	=3	CHARSTR	=6

Bytes 1- Value

EMPTY	unused (nonexistent)
BOOLEAN	7 zero-bits + 1-bit value (TRUE=1/FALSE=0)
INDEX	2-byte unsigned value
INTEGER	4-byte two's complement value
BITSTR	2-byte unsigned bit count N + bit string + zero padding to byte boundary
CHARSTR	2-byte unsigned character count N + ASCII string
LIST	2-byte element count N + element data objects

APPENDIX C—A DETAILED ENCODING OF THE PROCEDURE CALL PROTOCOL

Although the data types and transmission formats detailed in the previous appendixes serve primarily as vehicles for representing the arguments and results of remote procedures, they can just as readily and effectively be employed to represent the commands and responses by which those parameters are transmitted.

Taking this approach, one might model each of the two Protocol messages as a PCP data object, specifically a LIST whose first element is an INDEX message type. The following concise statement of the Protocol then results:

LIST	(CALL,	tid,	procedure,	arguments)
	INDEX=1	INDEX/EMPTY	CHARSTR	LIST
LIST	(RETURN,	tid,	outcome,	results)
	INDEX=2	INDEX	BOOLEAN	LIST

The RESULTS of an unsuccessful procedure would be represented as follows:

LIST	(error,	diagnostic)
	INDEX	CHARSTR

APPENDIX D—A LOOK AT SOME POSSIBLE EXTENSIONS TO THE MODEL

The result of the distributed-system-building strategy proposed in the body of this paper and the preceding appendices is depicted in Figure 3. At the core of each process is the inter-process communication facility provided by the operating system, which effects the transmission of arbitrary binary data between distant processes. Surrounding this core are conventions regarding first the format in which a few, primitive types of data objects are encoded in binary for IPC, and then the formats of several composite data objects (that is, messages) whose transmission either invokes or acknowledges the previous invocation of a remote procedure. Immediately above lies an open-ended protocol layer in which an arbitrary number of enhancements to the distributed programming environment can be implemented. Encapsulating these various protocol layers is the installation-provided runtime environment, which delivers DPS services to the applications program according to machine- and possibly programming-language-dependent conventions.

The Protocol proposed in the present paper recognizes only the most fundamental aspects of remote procedure calling. It permits the caller to identify the procedure to be called, supply the necessary arguments, determine the outcome of the procedure, and recover its results. In a second paper,[19] the author proposes some extensions to this simple procedure call model, and attempts to identify other common forms of inter-process interaction whose standardization would

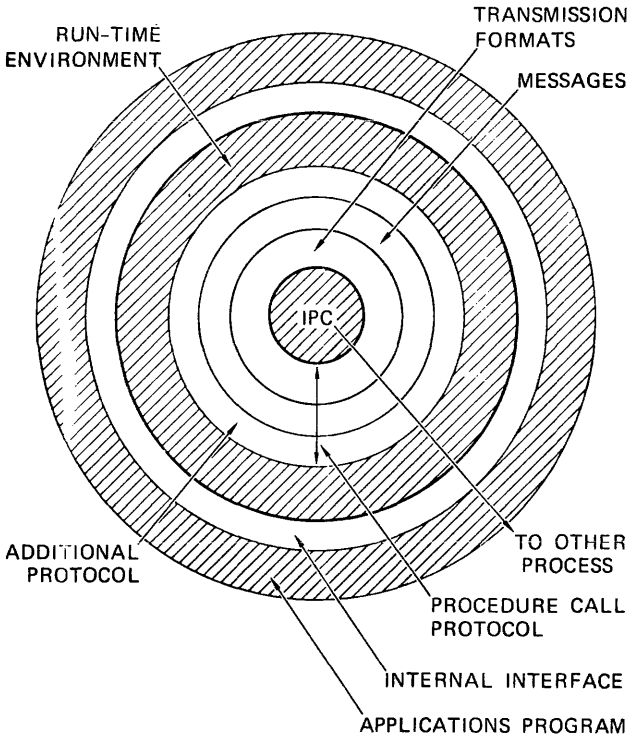

Figure 3—Software and protocol layers comprising a process within the distributed programming system

enhance the distributed programming environment. Included among the topics discussed are:

(1) Coroutine linkages and other forms of communication between the caller and callee.
(2) Propagation of notices and requests up the thread of control that results from nested procedure calls.
(3) Standard mechanisms for remotely reading or writing system-global data objects within another program.
(4) Access controls for collections of related procedures.
(5) A standard means for creating and initializing processes, that is, for establishing contact with and logging into a remote machine, identifying the program to be executed, and so forth. This facility would permit arbitrarily complex process hierarchies to be created.
(6) A mechanism for introducing processes to one another, that is, for superimposing more general communication paths upon the process hierarchy.

These and other extensions can all find a place in the open-ended protocol layer of Figure 3. The particular extensions explored in Reference 19 are offered not as dogma but rather as a means of suggesting the possibilities and stimulating further research.