Advanced program-to-program communication in SNA

by J. P. Gray
P. J. Hansen
P. Homan
M. A. Lerner
M. Pozefsky

Systems Network Architecture (SNA) defines the behavior of networks of heterogeneous, loosely coupled processors. This paper describes the development of program-to-program communication services in SNA and introduces Advanced Program-to-Program Communication (APPC), the culmination of this development. It also discusses the use of APPC in the construction of distributed services and shows that SNA with APPC and other SNA services can be thought of as a distributed operating system.

Systems Network Architecture (SNA) was announced by IBM in 1974. Since then, the original set of functions, which supported distribution of data processing between applications in a single central processor and multiple distributed cluster controllers, has been enhanced by the addition of many new functions and new products. This paper assumes some familiarity with SNA. (See References 1–14.)

The first section of this paper reviews growth trends in networks and distributed processing. The second section briefly reviews the overall design of SNA. After distinguishing between the physical network of nodes connected by data links and the network of Logical Units (LUs) connected by sessions, this review emphasizes the properties of LUs, sessions, and application programs.

The third section reviews applications within networks and discusses the requirements for interprogram communication that these applications place upon the architecture. The last section introduces Advanced Program-to-Program Communication (APPC) and shows how APPC meets many of these requirements.

Growth trends

The rapid improvements in computer and terminal function together with declining costs suggest continued large growth in applications during the rest of the 1980s. The number of installed workstations—many of them intelligent—may reach twenty million during the next five years. Because they are easy to develop, small (independent) applications for these workstations are expected to be available in wide variety. Applications that can exist only in interaction with other applications and remote data will be more difficult to develop and so will be available in less variety.

An important growth area is local area networks.[15,16] Relatively inexpensive direct connections are featured by local area networks, along with low delay and high bandwidth. The SNA design, which

keeps the logical network[17] of LUs and sessions independent of physical configurations, allows low delay, high bandwidth, as well as inexpensive connections and any other physical network properties to be exploited if they are present. The network owner can configure the physical network to meet cost, performance, installability, maintainability, and security requirements.[18-21] In this regard, SNA is similar to operating systems that are not limited to a specific machine configuration but which can operate over a wide range of processor, storage, and device configurations.

Lessons from an earlier period of growth in the computer industry can be applied to the current rapid growth in distributed processing.

Lessons from an earlier period of growth in the computer industry can be applied to the current rapid growth in distributed processing. When machines became large and reliable enough to support multiprogrammed operating systems, the importance of maintaining compatibility between different applications and different computer installations quickly became apparent. While some applications and some data were isolated, many programs and data were interrelated. Installations adopted a limited number of languages and put their data in shared files or data bases. Utility programs (e.g., sort) and other system services (e.g., spooling supervisors) were adopted in preference to application-specific programs. The savings that resulted were primarily in the most scarce of resources: skilled people.

The same forces are still at work; benefits will accrue from limiting the number of languages used in a network, from providing shared, secure access to the data around the network, and from adopting network utilities and services in preference to application-specific programs. The development and wide use of network utilities, the advent of intelligent workstations and small processors dependent on servers, and distributed applications, in general, require a set of architecturally defined interprogram communication primitives as a foundation.

Systems Network Architecture (SNA)

LUs and data transport. An SNA network consists of nodes connected by data links. Path control elements in each of these nodes route packets[22] from resource managers—called Logical Units (LUs)—along logical connections—called sessions—to their destination LUs.[23] The collection of all path control elements and the data link control elements that interconnect them constitute a transport network. Nodes can be connected by multiple links and contain a variable number of LUs.

An LU initiates a session with a partner by providing the partner's LU name. This LU name is transformed into an address placed into a packet header following access to a local or remote directory containing name-to-address translations.[24] Actual execution-time routing uses the addresses carried in the packet headers. A name that characterizes the mode of service to be provided by the session (e.g., "fast," "bulk," "secure") is also specified at session initiation.

A distributed operating system. An SNA network can be viewed from many perspectives. One of the more fruitful ones is to view it as a distributed operating system,[25-33] where the network is decomposed into programs or processes[34] running on a shared (distributed) operating system and connected by suitable interprogram communication. Of course, each program actually runs on a local operating system. Figure 1 illustrates two programs using their shared operating system to help them communicate.

Figure 2 illustrates two programs using SNA as their shared, distributed operating system to communicate. When the operating system is distributed, interprogram communication has to use message exchanges rather than shared storage.[35] Inasmuch as remote program communication is restricted to message exchange mechanisms, they should be used with local program partners to provide local/remote transparency. This means that, in most cases, communication between programs has the flavor of access to such I/O objects as files, rather than the flavor of access to data structures mapped into local memory.

Figure 1 Programs communicate with the aid of an operating system

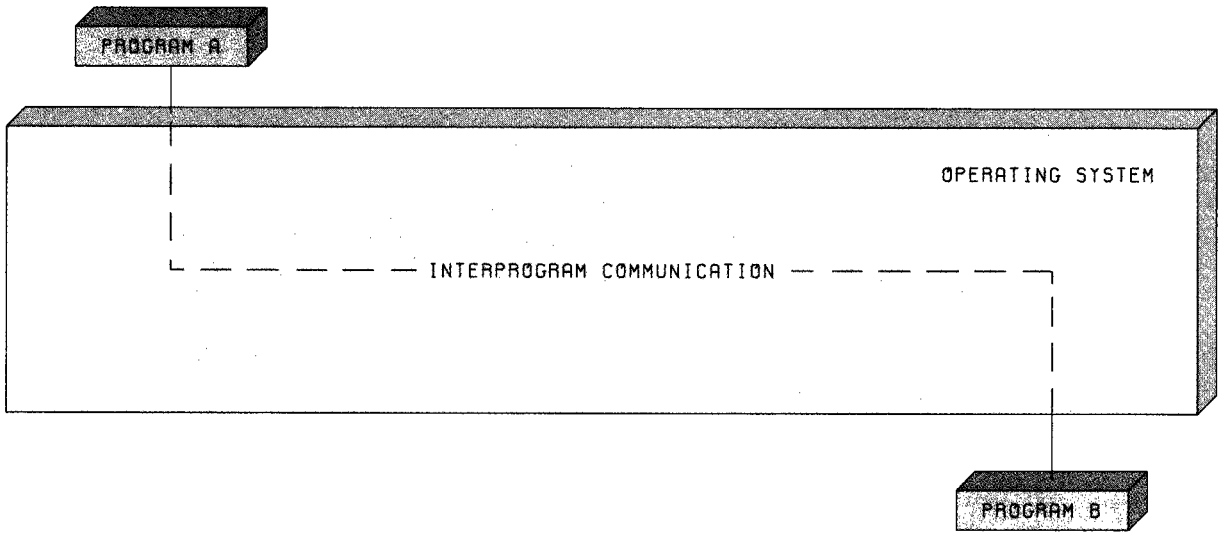
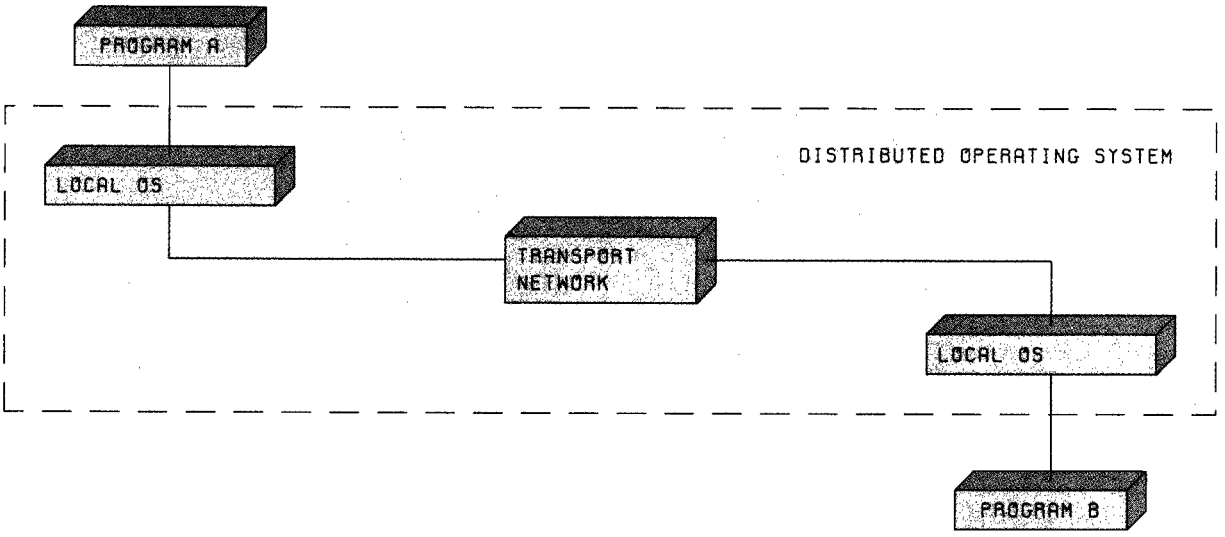
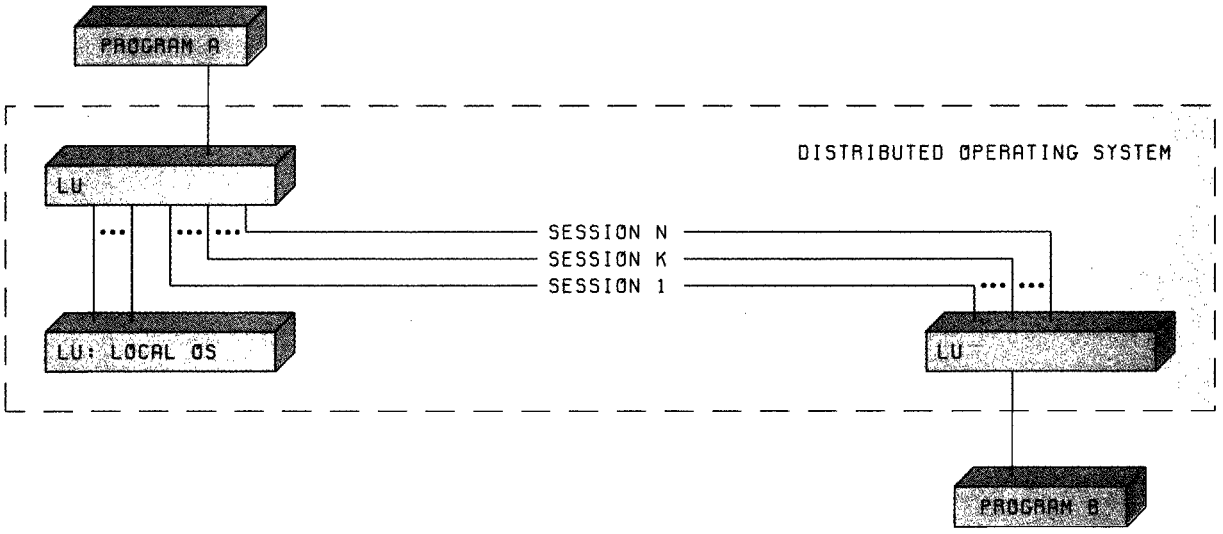


Figure 2 Distributed programs are served by local operating systems



The effects of inhomogeneity. We often conceive of an operating system as a designed structure, similar to a car or a painting. This conception, however, becomes less and less true as the size of the computer system being managed by the operating system increases. Most large systems are in a continual state of change, with repairs, hardware upgrades, and software upgrades all being interleaved with application processing. They are less designed than evolved. A distributed operating system, being larger, is yet more organic, with many elements at different (but compatible) levels of function. Even in the simplest case of a network of identical nodes with identical software, repair of

Figure 3 A network of sessions between LUs



software errors introduces inhomogeneity. This is the case because the network cannot (realistically) be changed instantaneously. Nor can a typical large network be shut down enough to change every node at the same time. It follows that network protocols must be capable of tolerating growth and change.

Similarly, network applications must also be capable of tolerating growth and anticipating change. Whereas the two halves of a distributed application may at first run on nodes known to the initial designers, one or the other half is likely to run on a different type of node in the future. Further, services (e.g., file access) that were local may be distributed later. All of this creates a requirement for a common set of remote services for application programs. This can be met in either of two ways: (1) all the nodes over which applications are to be distributed can be within the same product family (e.g., all might be CICS/VS, although not all at the same release and maintenance levels), or (2) the services can be provided by common, architecturally defined protocols. These two solutions are not mutually exclusive. A network can contain applications that are designed to take advantage of either or both of these types of distributed services.

The ability to distribute applications brings with it many design problems (e.g., how to divide a given function, how to divide data, how to recover from partial failures), but this ability also brings new opportunities. The partitioning that is required to distribute functions creates new and durable interfaces at which growth and evolution can occur. When functions are divided along correct lines, the pieces become building blocks to be used in future applications.

The development of distributed applications is expensive, a fact that creates a demand for transparently distributed services (e.g., file access) and for network utilities (e.g., file transfer). SNA products have answered this demand with a variety of distributed services and utilities. For examples, see References 36–46.

LUs as local operating systems. We now hide the transport network and concentrate on the network of LUs and sessions shown in Figure 3.[47] Just as adjacent nodes can be connected by multiple links, LUs can be connected by multiple sessions, called *parallel sessions*. The LUs are more than merely ports for message traffic: each LU provides operating system services (including interprogram communication) to one or more local programs. That is, each program sees an LU as being the local operating system on which it runs. The result is that each program sees the network of loosely coupled LUs connected by sessions as a distributed operating system.

Figure 4 The mapping of products into LUs

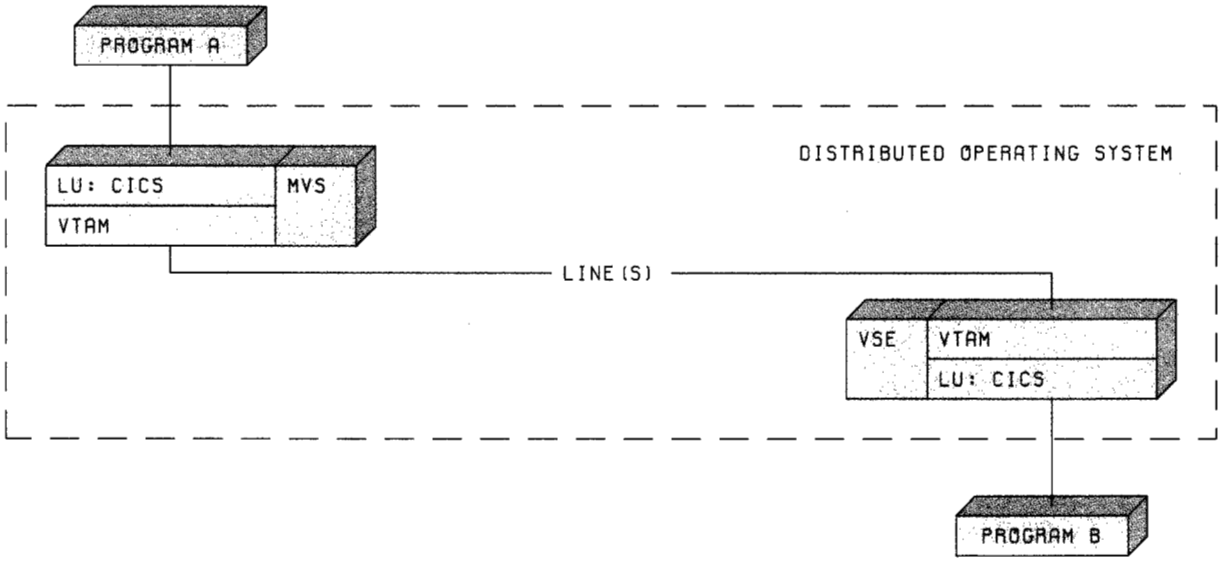


Figure 4 illustrates the relationship of LUs to products. In this example, the LUs are both CICS/VS, one running on MVS and the other on VSE. In both cases, VTAM implements the local portion of the transport network.[48] Other portions of the physical network, such as IBM 3725 multiplexors and modems, are lumped in with LINE(s) in the figure. Notice that there are multiple levels of interpretation in this example: the hardware interprets the microcode, which interprets MVS (or VSE), which interprets VTAM, which interprets CICS, which interprets the application program. Sometimes one interpreter completely hides a lower one. MVS does not see the hardware; it sees only the System/370 instruction set (ignoring diagnostic instructions). On the other hand, VTAM sees both MVS services and the System/370 instruction set. CICS might be described as running on MVS and using VTAM services, thinking of VTAM as an extension of MVS (which, as a privileged subsystem, it is). Not only alternative descriptions but also many varied implementations are possible. For example, the System/38 system structure includes the equivalent of VTAM under the machine interface. These variations, although interesting and important at each node in the network, are not exposed to other nodes in the network.[49] This makes it possible to define a single architecture rather than a unique interface between every pair of products.

LUs manage resources. Each LU makes a set of resources available to its programs. The exact set is product and configuration dependent; examples are the following: processor cycles, main storage, files on disk or tape, such I/O devices as keyboards or displays, and such abstract resources as sessions, queues, or data base records. Some of these resources are local to a program; that is, they are attached to the same LU as the program. Other resources are remote; that is, they are attached to other LUs. (The LUs might be within the same physical node.) Sessions are local resources at each LU, but they are shared between LUs. Most of the distributed utility services[50] and all the applications provided by an SNA network are provided by programs that run on LUs and use sessions to communicate among themselves. Terminals are not special in this regard. Fixed-function terminals have built-in programs that define the terminal's behavior.[51]

Resource allocation is a central function of the LUs. Programs can ask the LU for access to a resource. The LU then schedules access to its wholly-owned resources (such as files), coordinates the allocation of shared resources (such as sessions), and creates new copies of abstract resources, including sessions, when necessary. The larger, higher-function LUs may also provide resource allocation deadlock

detection, resource change commitment control, resource access security, and resource formatting (or presentation) services to their programs.

Sessions and conversations. The sessions carrying messages between LUs or programs running atop LUs are resources shared between those LUs, as shown in Figure 5. The first SNA products[52] used sessions between LUs to connect remote application programs to centralized application programs. Each session was dedicated to the use of a single program at each LU. Very quickly, however, the number of programs associated with an LU grew. Furthermore, any-to-any connectivity among the programs in each LU became the norm. Hence, the concept of the conversation was developed. A *conversation* is a serial time slice of a session.[53] This extension recognizes the large differential cost between activating a new session, which might involve many network components and the execution of many instructions, and activating a new conversation by using an existing, but not busy, session.

The resulting use of conversations has become recognizably that of *transactions*. Hence, application programs running on LUs are called *transaction programs*. Figure 6 illustrates the sharing of a conversation between two programs.

Two LUs connected by one or more sessions share responsibility in the allocation of sessions to transaction programs for use as conversations. When both LUs elect to allocate the same session resource, contention can occur. This condition is resolved by making one end of the session the contention winner and the other end the contention loser.[54] The LU that loses an allocation attempt tries again on another session, or if none is free, it activates an additional session. If this is not possible, the LU queues the request for a conversation until it can be satisfied. When a conversation ends, the session on which it was carried becomes free for reallocation.

LU types. In order to provide useful communication within a distributed operating system, the local operating systems (LUs) have to share a common set of protocols. These protocols have the following two functions: (1) activating a session between two LUs, which is analogous to establishing a telephone connection between two offices, and (2) using the session to communicate, which is analogous to the dialogue that two persons exchange over a telephone connection. Just as a caller establishes the language of the call when he says "hello" rather than "buenos días," a protocol called an *LU session type* (sometimes "LU session type X" is shortened to "LU X")[55] is established at the time a session is created. The LU types fall into three groups:

- *Not specified by SNA.* LU 0 is defined by implementations. A number of products have defined

Figure 5 Sessions are resources shared between LUs

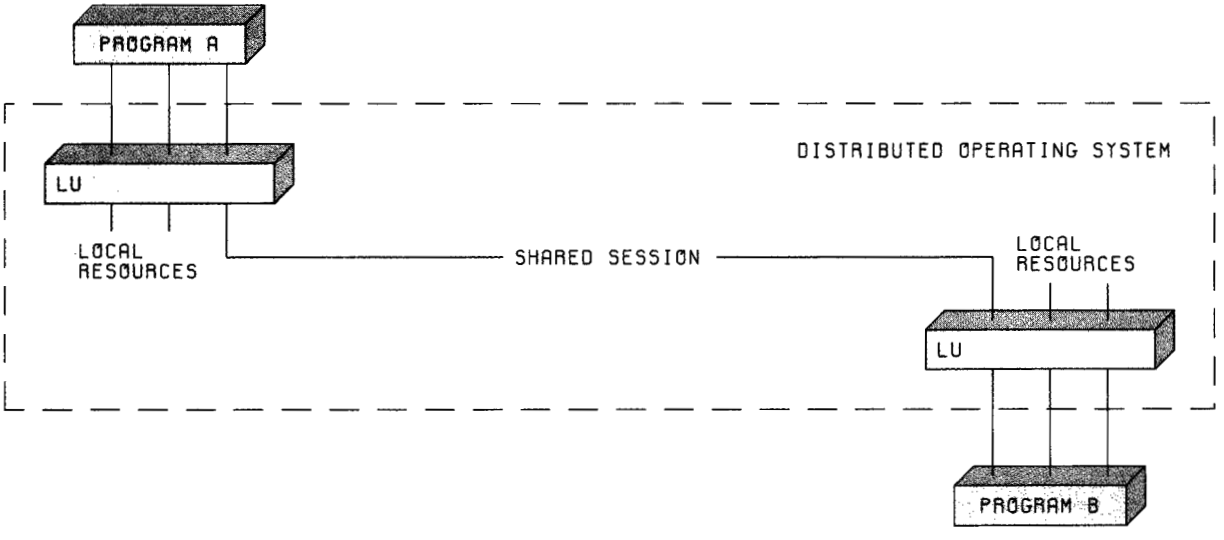
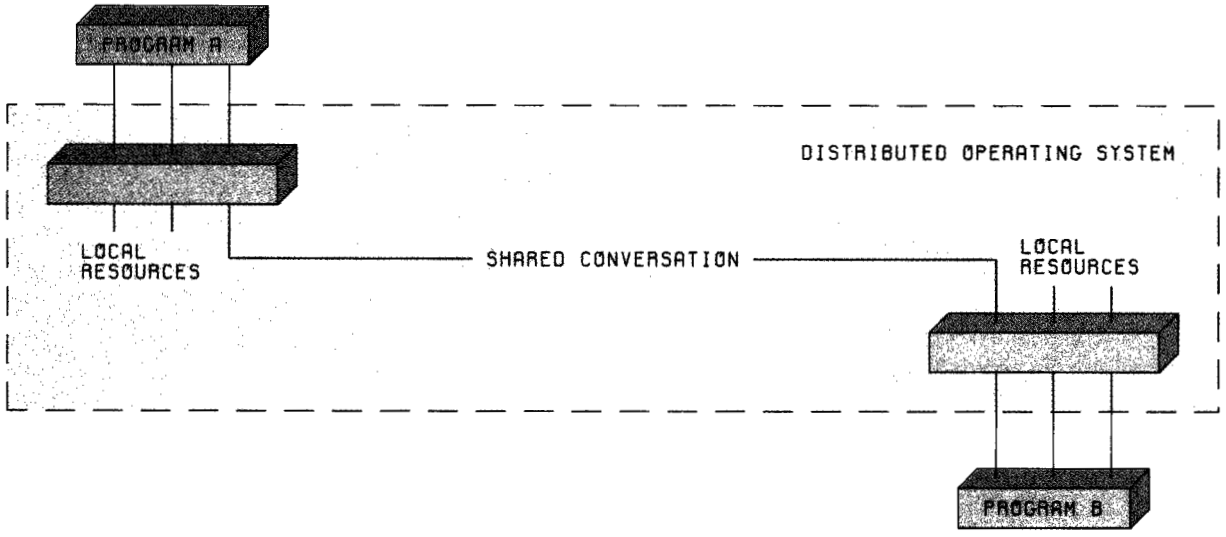


Figure 6 Transaction programs share conversations



their own program-to-program protocols. (See References 36, 38, and 56.)

- *Terminals*. The behavior of the terminal as seen by the host program is architecturally defined so that multiple terminal implementations can be supported at lower implementation costs to the host programs than would result from individualized support.

  LU 1. Applications use LU 1 to access such nondisplay I/O devices as printers and keyboard-printer terminals. The terminal is modeled as built-in transaction programs with local resources that can include keyboards, console printer, line printers, card punches, card readers, diskettes, and hard disks.

  LU 2. Applications use LU 2 to access display terminals with the IBM 3270 data stream.[57]

  LU 3. Applications use LU 3 to access printers with a subset of the IBM 3270 data stream.[58]

  LU 4. Applications use LU 4 to access terminals that are similar to LU 1 terminals.

  LU 7. Applications use LU 7 to access display terminals with the IBM 5250 data stream.
- *Program-to-program*. LU 6 provides SNA-defined interprogram communication protocols and is the base on which the IBM distributed operating system function is evolving.

The definitions of terminal LU types emphasize the terminal end of the session, with few constraints placed on the host end of the session. This allows optimizations of host programming support for these LU types. Sometimes this leads to the same functions being achieved by different means. For example, no constraints are defined on the mapping of terminal operator dialog-to-session protocols. Thus, one product might use several session-level conversations to accomplish the same thing another product does within a single conversation. Whereas the resulting dialog can look the same to an operator at a display terminal, the detailed message sequences exchanged over the session can be quite different. Terminal protocols and applications at the host often take advantage of the decision-making capability of the human operator who is thought to be present at the terminal. As a result, any attempt to use a terminal LU type as an interface between application programs has to deal not only with the problems created by the definitional emphasis on the terminal end of the session but also with the additional burden of simulating a human operator. Human flexibility, especially in error recovery situations, is difficult to imitate in programs.

To illustrate the specialized nature of terminal LU types, consider LU 2. This LU type has restrictive assumptions built into its error recovery protocols. The host LU is assumed to be responsible for all error recovery algorithms. Restrictive assumptions are also built into its resource naming. The terminal's LU name is implicitly the name of the display.

Such assumptions are also built into its security facilities. An operator at the terminal LU is assumed to be able to enter log-on information on the display. Similarly, such assumptions are built into output scheduling algorithms that are used by hosts. Output is limited to one display screen's worth of data so as not to overrun the operator's ability to read the display. Restrictive assumptions are also built into its IBM 3270 data stream.[5,58]

LU type 6 provides a general-purpose interprogram protocol that avoids the limitations of terminal LU types. Its latest release, LU 6.2,[59] is also referred to as Advanced Program-to-Program Communication (APPC).

Interprogram communication requirements

Interprogram communication protocols are used by a wide variety of applications and services. Here we discuss the more important of these requirements.

Abstraction. Programs using an interprogram communication service should not need to be aware of the details of how the service is implemented. This is a widely recognized principle of modularity—or layering—that has been applied in SNA from the beginning.

Semantic completeness. The primitives made available to the communicating programs should be complete in the sense of leaving out no needed functions. To give an obvious example, both the sending and receiving of data need to be supported functions. Less trivial are the following functions:

- Notification of conversation failure.
- Selection of transport characteristics.
  Delay. Interactive applications require very short response times, hence low transport delays, for best productivity.[60]
  Capacity. Batch applications require sufficient capacities to meet production schedules.
  Security. Sensitive data must be protected from unauthorized access or modification while being transported between programs.
  Cost. Some applications, especially transport of bulk data, must use the lowest-cost transport in order to be justified.
- For general usefulness, the protocols must be adequate for general-purpose application to general-purpose application. Fixed-function terminals are predefined applications running on (possibly) a subset of a general-purpose processor.
- To support multiple users, two adjacent LUs may have to support several concurrent conversations. This implies that multiplexing, as exemplified by parallel sessions, is required.[61]
- For security, access to such secure resources as data bases must be controllable.

To encourage wide use, the protocols should be easily incorporated into high-level languages.

- Commitment control requires that the LU must support checkpointing for distributed applications when desired, including resynchronizing after LU or session failures.
- Accounting requires that the communication protocols make transaction identifiers available for use in accounting, tracing, and activity logging.

Efficiency. Although ease-of-use is important, especially in reducing the cost of developing new applications, it is also necessary to provide a highly efficient protocol. Session-level exchanges should be equal to or fewer than the number of invocations of the program's interface to the session resource.

High-level languages. The communication primitives have to be available in a wide variety of products in basically equivalent semantic forms. To encourage wide use, the protocols should be easily incorporated into high-level languages. Mapping or formatting of the data that are sent and received must be supported.

Control functions. Adequate control over the total work load between any pair of LUs should be available to the controlling operators of the LUs involved. For example, if one operator wants to stop operation gracefully, the other operator must be informed so that one LU is not reactivating sessions as fast as the other LU is deactivating them.

Subsets. The protocols must be subsettable to allow implementation by the smallest products while still

supporting function adequate for the largest ones. This subsetting must be controlled to ensure the maximum feasible connectivity. In addition to a base of function required of all implementations, the optional functions must be collected into sets. An option set must be implemented totally or not at all. By limiting the number of option sets, the maximum amount of function is made available between given pairs of implementations.

Migration. LU 6.2 must support application programs that use earlier program-to-program protocols.

Defining Advanced Program-to-Program Communication

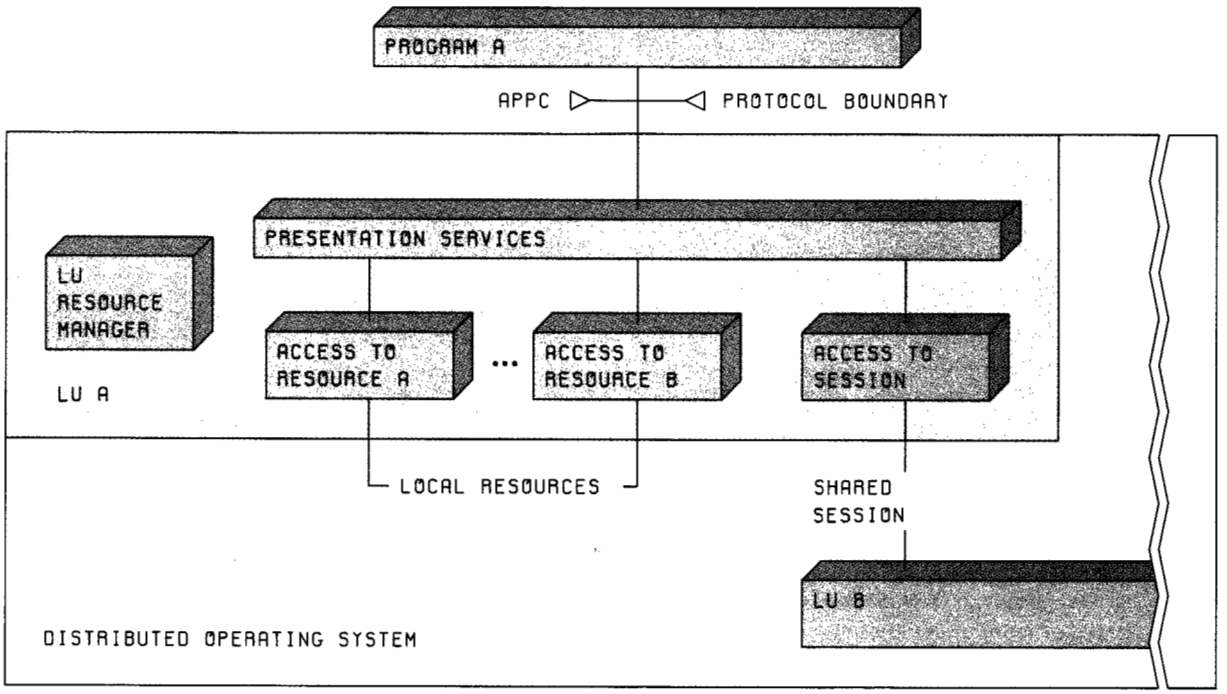
The requirements discussed in the preceding sections have been met by SNA's Advanced Program-to-Program Communication (APPC). We now describe APPC and discuss how the APPC design meets the requirements for an interprogram communication protocol.

Abstraction. The requirement to provide a high degree of abstraction in the definition of the APPC protocols had to be met within the context of the existing SNA definitions,[3] which are in the form of a canonical implementation of an SNA node.[62] This implementation model has grown as the functions and scope of SNA have evolved.

The method of defining the LU 6.2 conversation functions is in terms of programming-language-like statements, called *verbs*. Documentation with verbs, which are completely defined by the procedural logic that generates session flows, provides significantly greater precision than English prose. Figure 7 shows how the verbs define the interactions between transaction programs and LUs for conversation resources. A set of verbs is referred to as a protocol boundary rather than as an application program interface, in order to distinguish them from the functionally similar interfaces that products provide for the use of their application programs.[63]

The presentation services component interprets verbs and can be thought of as including a subroutine for each verb. The LU resource manager does allocation of conversation resources and assignment of conversations to the sessions, keeping queues of free sessions and pending allocation requests. Its

Figure 7 Programs communicate with the aid of an operating system



equivalent component in products also allocates local resources in product-specific ways.

Semantic completeness. The following verbs that define conversations provide a variety of functions.[64]

SEND_DATA moves data into a buffer and returns control to the transaction program. The conversation resource supports the sending of arbitrary amounts of data structured as a series of variable-length records delimited by two-byte length fields. The data are actually sent on the conversation either as a result of a subsequent verb (e.g., CONFIRM) or when the buffer is filled.

RECEIVE_AND_WAIT returns data and/or control information to the transaction program. Once issued, the transaction program is in a wait state until something arrives on the conversation.

PREPARE_TO_RECEIVE marks the end of a message and gives up the right to send to the partner program. PREPARE_TO_RECEIVE causes control information to be sent to the receiving program to inform it that it has send control.

FLUSH causes all buffered data and control information to be sent.

REQUEST_TO_SEND asks the partner program for the right to send. This interrupt capability does not require truly asynchronous reporting of the REQUEST_TO_SEND. The interrupt notification occurs as a return code on a verb issued against the conversation.

SEND_ERROR reports an error in the data being received by terminating the incoming message,[65] purging the pieces that are buffered in the conversation, notifying the sending transaction of the error, and reversing the flow so that the program that issued SEND_ERROR obtains send control. With send control, the transaction program that issued SEND_ERROR can optionally use a data transfer to convey further information about the error and trigger whatever error recovery the transaction programs are designed to support.

Similarly, the sending program may detect an error in its local resources that makes it impossible to complete the message it is sending. The sending program then issues SEND_ERROR while retaining send control, and the receiving program is notified of the error condition.

CONFIRM ends a message and asks the partner program for assurance that no errors have been detected in it. The receiving program can reply with CONFIRMED if it has not detected any errors, or it can issue SEND_ERROR.

ALLOCATE spawns new activity at another LU by building a conversation to a named partner program. The named partner is placed in execution and given addressability to the conversation that started it. Thus, the ALLOCATE verb carries several parameters, including the following:

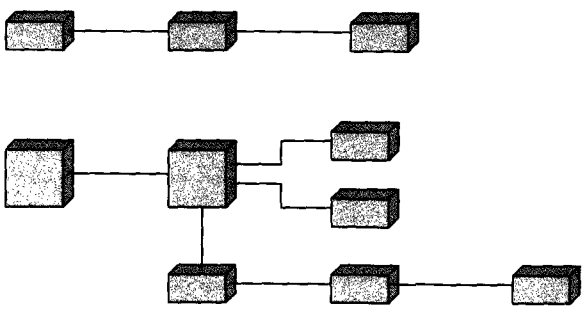
- LU_NAME is the name of the LU at which the partner program is located.[66]
- TPN is the Transaction Program Name of the partner program with which the conversation is desired.
- MODE_NAME specifies the type of transportation service that the conversation is to provide. For example, a SECURE, a BULK, or a LOW_DELAY conversation can be requested. The LU uses a session with the appropriate MODE_NAME to carry the conversation.[67]

The dynamic allocation of resources creates the possibility of resource allocation deadlocks among transactions.

The target of the conversation is a newly created process or task,[68] which means that the distributed processing in the network at any instant of time consists of a number of independent, distributed transactions, each of which consists of two or more transaction programs connected by conversations. Graphs of sample distributed transactions are shown in Figure 8.

The dynamic allocation of resources creates the possibility of resource allocation deadlocks among the transactions at one LU or among transactions at several LUs. The designers of transactions have to consider this problem. For local resource deadlock,

Figure 8 Two examples of distributed transaction programs



they can either rely upon the deadlock detection services that the LUs may provide, or they can design every transaction to use the same order of allocation for all resources. Because deadlock detection can become complex when trying to detect possible deadlocks involving remote transaction programs,[69] distributed deadlocks—if they occur—are detected by timers.

DEALLOCATE ends the conversation. Inasmuch as either partner may issue DEALLOCATE, conversations vary from a single short message to many exchanges of long or short messages. For efficiency, the FLUSH, CONFIRM, and SYNCPT functions may be combined with deallocation to minimize session flows. A conversation could continue indefinitely, terminated only by a failure of an LU or by the session that carries it. Transaction programs are not ended by DEALLOCATE, but continue until they terminate their own execution, end abnormally, or are terminated by control operator action.

SYNCPT (syncpoint) makes the accumulated changes to multiple resources permanent. Also included in this function is the ability to abort. That is, the application can choose to roll back to the boundary defined by the previous syncpoint verb execution.

Committing changes atomically with SYNCPT is an optional service within APPC. Additionally, this function provides for recovery at the boundaries defined by the syncpoint verbs when connectivity is broken.

Conversations are defined to the syncpoint service in each LU at ALLOCATE time as either being protected by syncpoint or as being unprotected. In the latter case, the transaction programs are themselves responsible for error recovery synchronization.

The transaction programs have direct control over their use of the APPC verbs. Some verbs cannot be issued in certain states. For example, a program cannot issue SEND_DATA when it is not in send state. These restrictions enforce the SNA half-duplex and error notification protocols. Any additional protocol features that may be desired are created and enforced by the transaction programs. For example, they might obey the rule that only the program that starts a conversation may end the conversation.

Efficiency. Conversations are mapped efficiently onto sessions. For example, that which the programs see as two short messages (perhaps an inquiry and its reply) results in two short messages flowing in the network. Figures 9 and 10 are two examples of coupled transaction programs that illustrate the efficiency with which verb sequences are mapped onto session flows. These examples have been simplified by omitting some parameters from the verbs. Details may be found in Reference 1.

The verbs in Figure 9 have the following effects:

1. TP(a) issues ALLOCATE to request a conversation with partner program b. The LU creates an allocation request, using information provided in the ALLOCATE verb. The LU places the allocation request in the send buffer and returns control to TP(a), with the conversation in the send state. Nothing is sent.

2. TP(a) issues SEND_DATA, which causes the LU to place the data in its buffer behind the allocation request. The data are short enough that nothing is sent.

3. TP(a) issues DEALLOCATE with TYPE(FLUSH), which means that the deallocation is to be immediate. The LU sends the contents of its buffer with a DEALLOCATE indication. The conversation is now completed at TP(a). The session flow, consisting of one message, starts TP(b), with the conversation in receive state.

4. TP(b) issues RECEIVE_AND_WAIT and receives all the data.

5. TP(b) issues another RECEIVE_AND_WAIT and receives the DEALLOCATE_FLUSH indication.

Figure 9 A one-way conversation

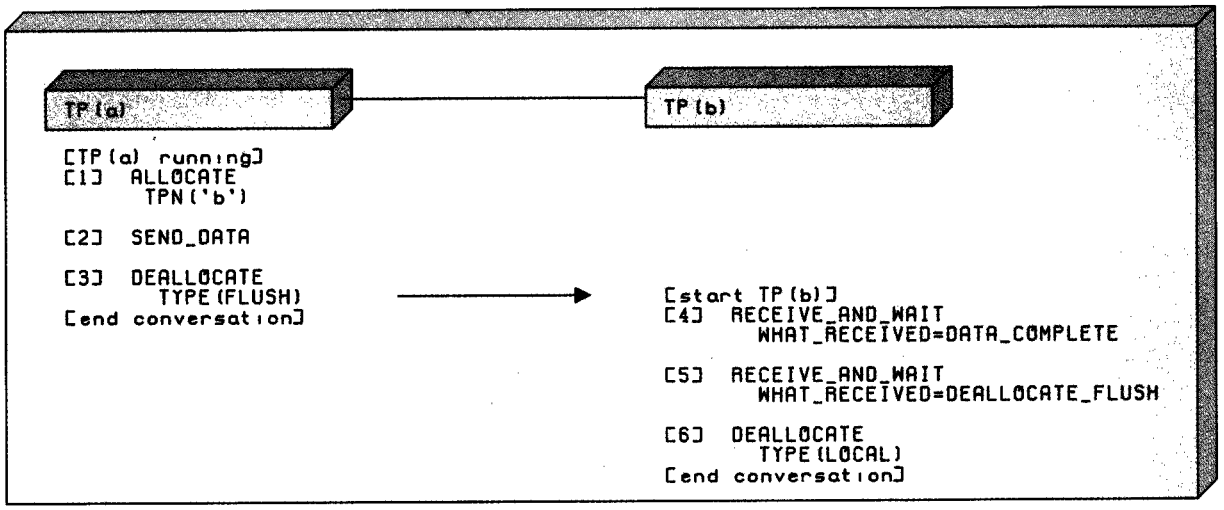
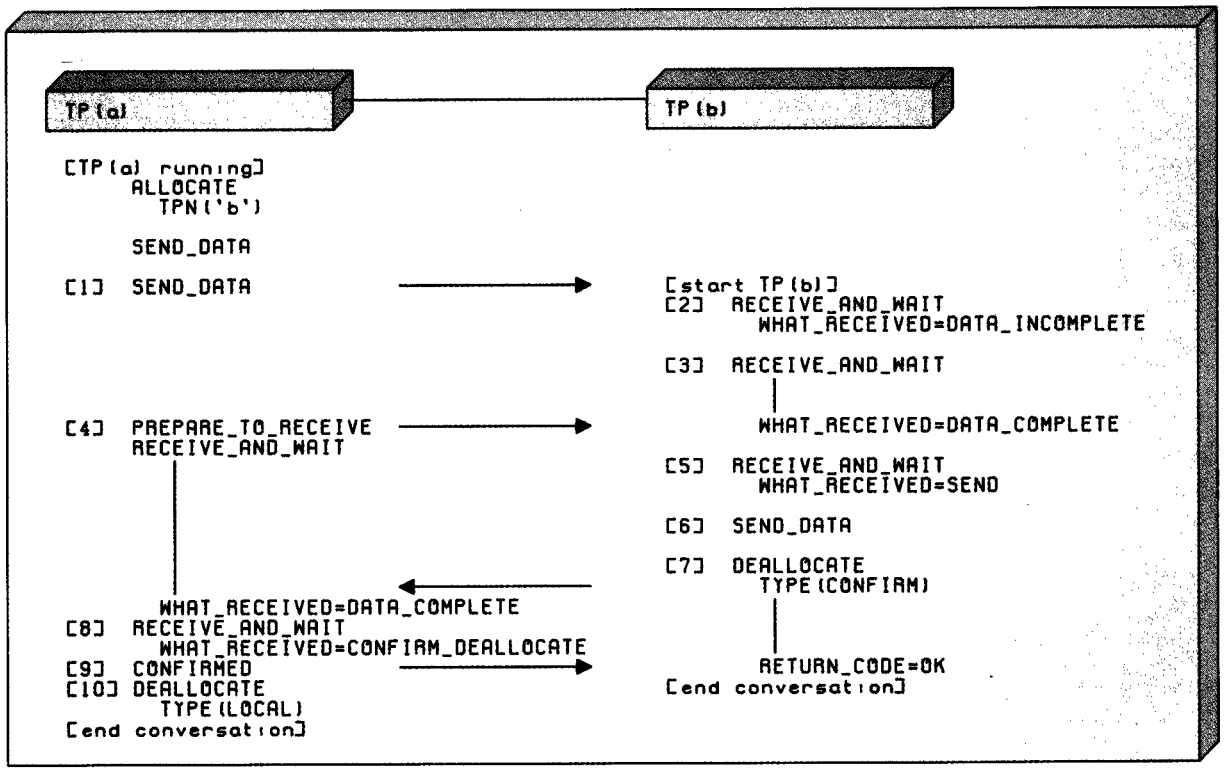


Figure 10 A two-way conversation with confirmation



6. TP(b) issues DEALLOCATE with TYPE(LOCAL), causing the LU to discard its control information for the conversation. This ends the conversation for TP(b). Both TP(a) and TP(b) continue in execution until they end themselves.

Notice that three verbs have been compressed into one message. If the message is short enough, it can flow as one packet.

Figure 10 shows additional features of the APPC verbs:

1. The first two verbs are the same as in the previous example, except that this TP(a) must send a larger amount of data. These data are long enough that the LU has to send some data while retaining a portion of the data in its local send buffer.

Performance of a distributed transaction is affected by the internal performance of the executing nodes.

2. TP(b) issues RECEIVE_AND_WAIT, thereby obtaining the first portion of the data.

3. TP(b) issues RECEIVE_AND_WAIT again, thereby causing the LU to suspend the execution of TP(b) until the remaining portion of the data has been received by the LU.[70]

4. TP(a) issues PREPARE_TO_RECEIVE followed by RECEIVE_AND_WAIT, which causes the LU to send the contents of its buffer together with the SEND indication. The execution of TP(a) may have been delayed because of the execution of other programs at its LU, because it may have been doing other processing that did not result in activity on this conversation, or because it may have executed the PREPARE_TO_RECEIVE verb immediately after the SEND_DATA, but the data may have encountered some delay in transit. The LU suspends execution of TP(a) until it receives data to satisfy the RECEIVE_ AND_WAIT.

   Control is returned to TP(b) as soon as the remaining portion of data is received by its LU.

5. TP(b) issues another RECEIVE_AND_WAIT and receives the SEND indication.

6. TP(b) issues SEND_DATA, causing the LU to place the data in its buffer. Nothing is sent.

7. TP(b) issues DEALLOCATE with TYPE(CONFIRM), which implies confirmation processing and causes the LU to send the contents of its buffer together with a CONFIRM_DEALLOCATE request. The CONFIRM causes the LU to suspend execution of TP(b) processing until it receives an affirmative or negative response.

   The LU returns control to TP(a), indicating that the program has received all the data.

8. TP(a) issues another RECEIVE_AND_WAIT and receives the CONFIRM_DEALLOCATE request.

9. TP(a) responds affirmatively by issuing CONFIRMED, thus causing its LU to send an affirmative response. A SEND_ERROR can be issued instead of CONFIRMED, in which case the conversation remains allocated at both programs.

   The LU returns control to TP(b) to indicate successful completion of the DEALLOCATE. The conversation is complete for TP(b).

10. TP(a) issues DEALLOCATE with TYPE(LOCAL), which causes the LU to discard its control information for the conversation. The conversation ends for TP(a).

Performance of a distributed transaction is affected by many variables, including the internal performance of the nodes at which each component transaction program executes. In smaller processors, it may be important to minimize buffer occupancy and keep the path lengths to send and receive packets low. In larger processors, it may be important to reduce the number of times transaction programs are dispatched (e.g., processor pipelines and caches drain when a new program is dispatched) by using large data areas even at the expense of some

increase in the directly measured path length per packet. In this way, total throughput and average path lengths for completion of a transaction can be improved.

With APPC, performance can be tuned by adjusting session packet sizes and numbers of available buffers. These changes do not affect the APPC verbs. With enough receive buffers, a transaction program can be dispatched only once for each message that it receives. Alternatively, if buffers are the critical resource, a transaction program can be dispatched once for every packet that is received.

With enough receive buffers, a transaction program can be dispatched only once for each message that it receives.

Another way in which APPC improves performance is by mapping conversations directly onto sessions. The resulting short path lengths preserve the efficiency of higher-level services.[71] For example, the Asynchronous SNA Distribution Service[72] moves data directly from the network into its final data set without moving it through a spool file.

High-level languages. So far, we have discussed basic conversations, those that provide full access to the communication primitives and complete control over the format of the transmitted data. APPC also defines a set of mapped conversation verbs (implementable with the basic conversation verbs) that hide certain options and details of the basic conversation verbs from the program. Mapped conversations are designed to be used by application programs written in high-level languages.

A major characteristic of a high-level language is that programs written in it are independent of the external representations of the data structures on which they operate. The mapped conversation verbs define optional support for data mapping operations, similar in concept to the familiar formatted I/O defined in languages such as FORTRAN and PL/I. When this option is being used, a map name is sent with the transmitted data so that the receiving map support can understand the format of the data.

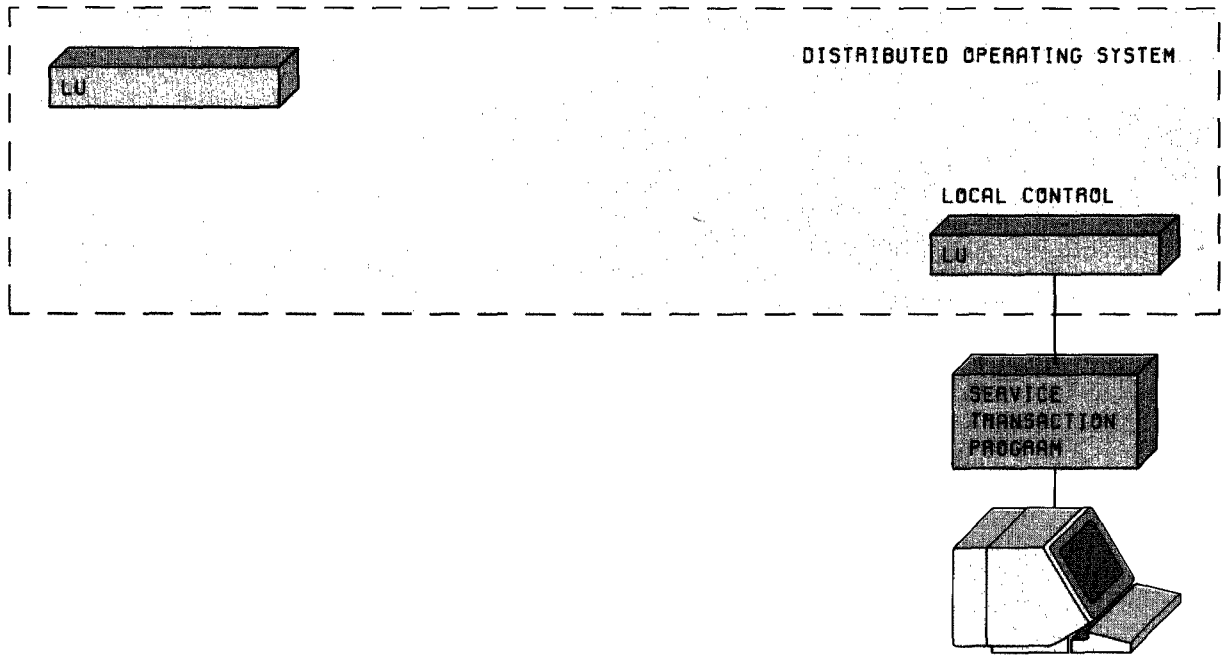
The set of mapped conversation verbs closely parallels the basic conversation sets and includes the following: MC_SEND_DATA, MC_RECEIVE_AND_WAIT, MC_PREPARE_TO_RECEIVE, MC_FLUSH, MC_SEND_ERROR, MC_CONFIRM, MC_CONFIRMED, MC_REQUEST_TO_SEND, MC_ALLOCATE, and MC_DEALLOCATE.

Control functions. Both network application programs and service transaction programs use the execution services provided by LUs. As shown in Figure 11, service transaction programs run on LUs in the same way as other transaction programs. They interact with a human operator, or they may run as a pure programmed operator. Many service transaction programs affect only the local LU. An example is the command to display the current set of active transaction programs.

Other control transactions, especially those that relate to sessions, can affect other LUs as well as applications at other LUs. For example, a local command to prematurely terminate a transaction that is using a conversation causes the conversation to be ended abnormally, a state change that must be transmitted to the partner LU for presentation to the transaction program that is sharing the conversation. Or a decision to deactivate one or more of the sessions shared by two LUs may be made by one LU's operator but must be communicated to the other LU. APPC includes several control operator verbs that provide LU-to-LU control and coordination, especially for activation and deactivation of sessions. This is illustrated in Figure 12.

When a distributed service transaction program starts at one LU, it creates a conversation to a partner transaction program in a partner LU. The two transaction programs then cooperate to perform the desired control activity. Error recovery logic handles such situations as operators attempting conflicting operations at the same time and session failures that can occur in the middle of a control transaction. Some of the APPC control operator verbs are the following:

Figure 11 Service transaction programs



INITIALIZE_SESSION_LIMITS determines the limits on the number of parallel sessions per mode name between two LUs. By agreeing on these limits in advance, the LUs can activate sessions with the partner's predetermined cooperation. This knowledge simplifies recovery from errors that may occur during attempts to activate sessions, ranging from mismatched system definitions to network failures.

RESET_SESSION_LIMITS resets to zero the agreed-upon session limits and also deactivates the sessions for a given mode name to a partner LU. Options allow queued requests for conversations to be satisfied before the reset is completed.[73]

ACTIVATE_SESSION activates one or more sessions for a given mode name.

DEACTIVATE_SESSION deactivates a specific session. Unlike RESET_SESSION_LIMITS, it does not change the session limits.

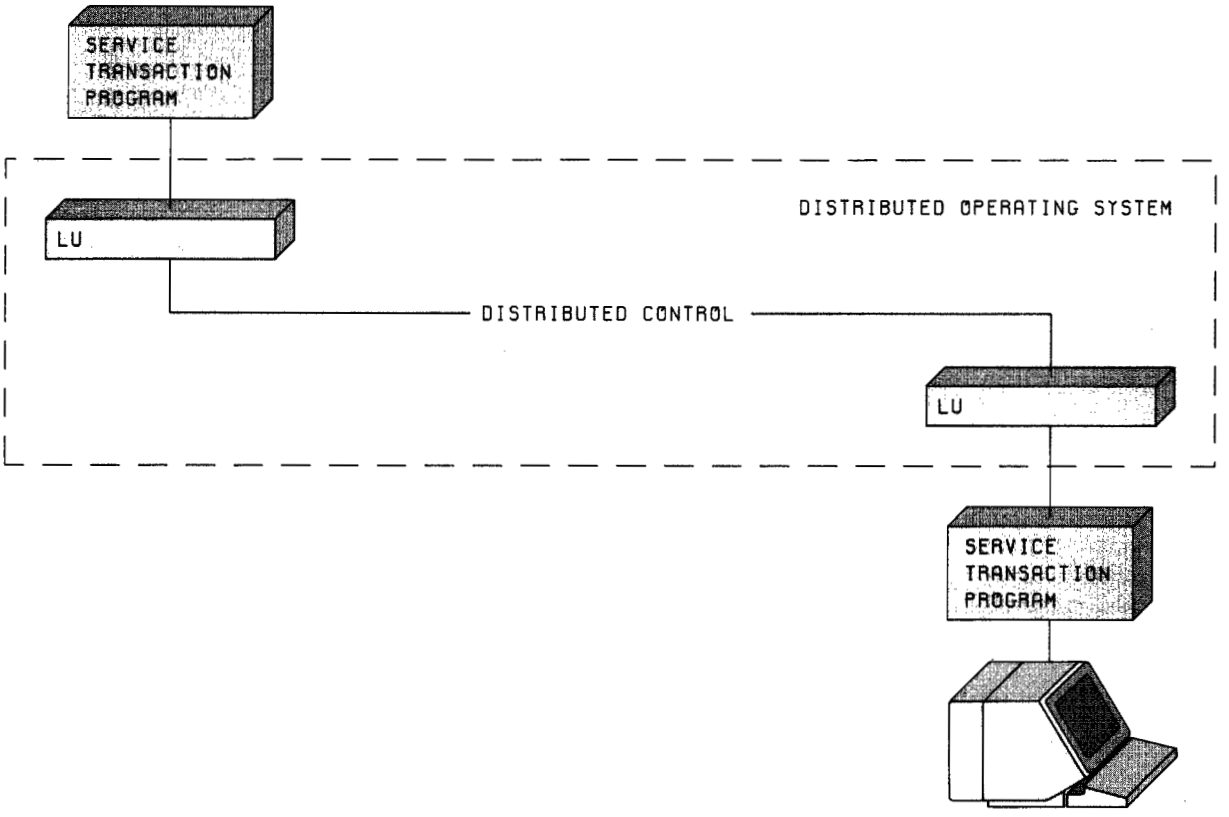
Control operator transactions are not the only LU-to-LU control flow in LU 6.2. Various data in the session activation command are used to reduce the amount of system definition required for two LUs to communicate. For example, at the time each session is activated, the LUs agree upon the maximum size of the packets that they will exchange.[74]

When the syncpoint functions are used, the LUs must ensure that compatible recovery logs are active on each LU, and they must also exchange resynchronization data after failures of conversations that were protected by syncpoint. Both of these activities utilize service transactions distributed between the partner LUs.

Subsets. The *SNA Format and Protocol Reference Manual*[3] describes SNA by defining, for example, with programming language declarations, the formats of messages that flow between network entities and the programs that generate, manipulate, translate, send, and receive those messages.

The *SNA Transaction Programmer's Reference Manual for LU Type 6.2*[1] defines the verbs that describe the functions being provided by the implementing products. Figure 13 illustrates a functional definition given in the programmer's reference manual. Not all the parameters on ALLOCATE are shown.

Figure 12 Distributed service transaction programs

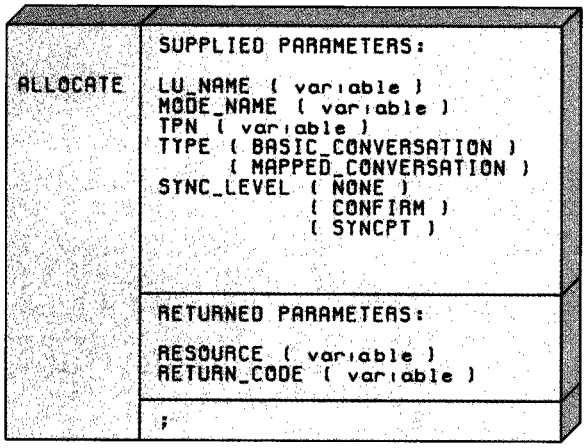


Although the meta-implementation technique explicitly and unambiguously defines all possible message flows, it does not illustrate which flows products must support (both on the send and receive sides) and which they may leave out. Furthermore, while defining the appropriate subsets of support is clearly the most difficult problem, conveying those conclusions in a written, unambiguous form is not simple. Defining supported functions using a programming language as noted earlier in the section on abstraction has the added benefit of providing a framework within which statements of required versus optional product support can be discussed. Each verb or parameter is specified individually as being in the base support required of all products or in one or more option sets, for both local and remote support.[1] For example, the ALLOCATE verb defined in Figure 13 is further defined in Figure 14.

The B's in Figure 14 indicate that support for the verb or parameter is in the base. The M's show that support is part of the mapped conversation option set. The numeral 1 shows that support is part of the syncpoint option set. The dashes show that returned parameters are not visible to the remote transaction program.

The APPC functions, represented as verbs and parameters, can be viewed in a functional relationship as well as in a subset-control relationship. Some functions can be used to implement other functions. This is, some functions are primitives from which other functions can be implemented. For example, ALLOCATE is used as part of error recovery for the syncpoint function. That is, after a session outage, the LUs run service transaction programs to exchange resynchronization information. Other portions of the syncpoint function use session encodings that are not present in the base. In this view, the base does not provide all primitives necessary to implement some of the options. To add the syncpoint option, an LU must provide additional pro-

Figure 13 Part of the ALLOCATE verb definition



gram support below the verb interface. Some option sets are built entirely on the base. Mapped conversations are an example of this.

The subset control and generic (architecture) description of functions provided by APPC permit the design and coding of distributed transaction programs without regard for the particular products upon which the individual programs are to be executed. The subsets required to support each of the distributed programs are clear to the designer, as a result of the verbs and parameters used for each distributed program. The subsets can be used to determine the set of products needed to support the required functions. Once these products are selected, the architecturally defined verbs can be translated in a straightforward manner to the particular languages supported by each of the products. If the designer limits himself to using only functions defined to be in the base, he is assured of the ability to implement the distributed programs on all LU 6.2 products, including those that may be added to the network at a later time. Since all optional functions are grouped into a limited number of option sets and since a product implementing any function in an option set must implement all functions in that option set, connectivity is also ensured between distributed programs (using a given option set) on different product implementations of LU 6.2 that support the option set.

Migration. As the latest release of LU 6, LU 6.2 has provided for the migration of most LU 6.0 and LU 6.1 application programs without change to the applications.[75] The LU 6.1 protocols closely correspond, from an application's viewpoint, to a subset of the mapped conversation verbs. Product publications should be consulted to determine the exact degree of compatibility. The LU 6.1 session encodings, on the other hand, are not an exact subset of the LU 6.2 encodings. The LU 6.1 encodings are not a subset of the LU 6.2 message encodings.

Concluding remarks

The requirements for interprogram communication have been shown to lead to Advanced Program-to-Program Communication (APPC), a shared resource environment specifically designed to support SNA's evolution as a distributed operating system.

SNA's design in general and APPC in particular provide a foundation upon which additional distributed processing services can be provided by IBM, other suppliers of hardware and software, and owners of individual networks. One example of such a service is described in the article "SNA Distribution Services" in this issue of the *IBM Systems Journal*. Another example is Document Interchange Architecture,[6] implemented by SCANMASTER, DISPLAYWRITER, and DISOSS. Past experience with services for local operating systems leads us to expect many more distributed services to come.

Acknowledgments

Many persons have contributed to the architecture and implementation of SNA. The following were chiefly responsible for the LU 6.0 and LU 6.1 architecture: Julian Jones, Dave Eade, Pete Lupton, and Pete Homan from CICS; Ed Cobb and Mike Stewart from IMS; Pete Hansen from System/38; Jim Gray, Mike Lerner, and Ron Ramos from Communication Systems Architecture. The following have been chiefly responsible for the LU 6.2 architecture: Pete Homan, John Cole, Phil Mead, Pete Lupton, and Roger Cath from CICS; Pete Hansen from the DDP Center in Rochester, Minnesota; Wayne Duquaine from System/34; John Fetvedt from System/38; Joel Webb from Series 1; Lynne Brooks from the ISC Project Office in Kingston, New York; Ed Cobb from IMS, Bob Nelson from Austin, Texas; Jim Gray, Mike Lerner, Mark Pozefsky, Ray Bird, Marsha Ferree, John Wilder, and Joe Austin from Communication Systems Architecture. Invaluable management coordination and encouragement have

Figure 14 ALLOCATE verb base and options definition

VERB AND PARAMETER	LOCAL SUPPORT	REMOTE SUPPORT
ALLOCATE	B	B
LU_NAME	B	B
MODE_NAME	B	B
TPN	B	B
TYPE(BASIC_CONVERSATION)	B	B
TYPE(MAPPED_CONVERSATION)	M	M
SYNC_LEVEL(NONE)	B	B
SYNC_LEVEL(CONFIRM)	B	B
SYNC_LEVEL(SYNCPT)	1	1
RESOURCE	B	--
RETURN_CODE	B	--

been provided by Bob Sundstrom, Terry Rogers, Tony McNeill, Diana Froelich, Robert Lee, Ken Coleman, Pete Hansen, Bob Chappuis, Ed Sussenguth, John Broughton, John Rood, and Nick Temple.

Cited references and notes

1. *Systems Network Architecture: Transaction Programmer's Reference Manual for LU Type 6.2*, GC30-3084, IBM Corporation; available through IBM branch offices.
2. *An Introduction to Advanced Program-to-Program Communication*, GG24-1584, IBM Corporation; available through IBM branch offices.
3. *Systems Network Architecture: Format and Protocol Reference Manual: Architecture Logic*, SC30-3112, IBM Corporation; available through IBM branch offices.
4. *Systems Network Architecture: Sessions Between Logical Units*, GC20-1868, IBM Corporation; available through IBM branch offices.
5. *Systems Network Architecture: Introduction to Sessions Between Logical Units*, GC20-1869, IBM Corporation; available through IBM branch offices.
6. *The Office Information Architectures: Concepts*, GC23-0765, IBM Corporation; available through IBM branch offices.
7. *Systems Network Architecture: Concepts and Products*, GC30-3072, IBM Corporation; available through IBM branch offices.
8. *Systems Network Architecture: Technical Overview*, GC30-3073, IBM Corporation; available through IBM branch offices.
9. J. P. Gray and T. B. McNeill, "SNA multiple-system networking," *IBM Systems Journal* **18,** No. 2, 263–297 (1979). This paper focuses on SNA's network services, as distinct from its operating system services.
10. D. P. Pozefsky and F. D. Smith, "A meta-implementation for Systems Network Architecture," *IEEE Transactions on Communications* **COM-30,** No. 6, 1348–1355 (1982).
11. G. D. Schultz, D. B. Rose, C. H. West, and J. P. Gray, "Executable description and validation of SNA," *IEEE Transactions on Communications* **COM-28,** No. 4, 661–677 (1980).
12. J. P. Gray, "SNA operating system services to support distributed processing," *Proceedings of the 1982 IEEE International Large Scale Systems Symposium*, 161–165 (1982). IEEE order number 82CH1741-8; available through IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854.
13. T. Schick and R. F. Brockish, "The Document Interchange Architecture: A member of a family of architectures in the SNA environment," *IBM Systems Journal* **21,** No. 2, 220–244 (1982).
14. R. J. Sundstrom and G. D. Schultz, "SNA's first six years: 1974–1980," *Proceedings of the Fifth International Conference on Computer Communication, Atlanta, Georgia, 27–30 October 1980*, 578–585 (1980).
15. R. C. Dixon, N. C. Strole, and J. D. Markov, "A token-ring network for local data communications," *IBM Systems Journal* **22,** No. 1–2, 47–62 (1983).
16. J. A. Saltzer, "Why a ring?" *Proceedings 7th Data Communications Symposium*, 211–217 (1981).
17. From an abstract point of view, a network is merely a set of nodes and edges connecting them. One can view an installed data communications network at several different levels of abstraction. If, for example, the abstract nodes are physical products and the edges are cables and communication lines, a *physical network* view is obtained. If the abstract nodes are LUs and the edges are sessions, a *logical network* view is obtained. If the abstract nodes are application programs and the edges are conversations (as discussed later in this paper), a *distributed application network* view is obtained.
18. SNA supports several local communication facilities today, including the SDLC loop on the IBM 8100,[19] the IBM 4331 systems, and up to 3-megabyte-per-second speeds between host processors running VTAM[20] connected with the IBM 3088.[21]
19. *IBM 8100 Information System Communication and Loop Description*, GA27-2883, IBM Corporation; available through IBM branch offices.
20. *ACF/VTAM General Information: Introduction*, GC27-0462, IBM Corporation; available through IBM branch offices.
21. *IBM 3088 Multisystem Channel Communication Unit Product Description Manual*, GA22-7081, IBM Corporation; available through IBM branch offices. Up to eight processors can be fully interconnected by one IBM 3088.
22. The packets in SNA are called Path Information Units (PIUs).
23. In addition to LUs, SNA defines *Physical Units* (PUs) and *System Services Control Points* (SSCPs) as senders and

receivers of messages. For more information, consult References 3, 7, 8, and 9.

24. The SSCPs in the network cooperatively provide an LUNAME-to-LU address directory and other related session services for use by LUs when sessions are being activated and deactivated. For details see References 3, 8, and 9.
25. G. D. Schultz first applied the distributed operating system paradigm to SNA in conversations during 1973.
26. P. H. Enslow, "What is a 'distributed' data processing system?," *Computer* **11,** No. 1, 13–21 (January 1978).
27. R. Eckhouse, J. Stankovic, and A. van Dam, "An overview of two workshops on distributed processing," *Computer* **11,** No. 1, 22–26 (January 1978).
28. H. C. Forsdick, R. E. Schantz, and R. H. Thomas, "Operating systems for computer networks," *Computer* **11,** No. 1, 48–57 (January 1978).
29. R. W. Watson and J. G. Fletcher, "An architecture for support of network operating system services," *Computer Networks* **4,** 33–49 (1980).
30. J. Mitchell and J. Dion, "A comparison of two network-based file servers," *Proceedings of the Eighth Symposium on Operating System Principles, SIGOPS* **15,** No. 5, 45–46 (December 1981).
31. R. Rashid and G. Robertson, "Accent: A communication oriented network operating system kernel," *Proceedings of the Eighth Symposium on Operating System Principles, SIGOPS* **15,** No. 5, 64–75 (December 1981).
32. A. Birrell, R. Levin, R. Needham, and M. Schroeder, "Grapevine: An exercise in distributed computing," *Proceedings of the Eighth Symposium on Operating System Principles, SIGOPS* **15,** No. 5, 178–179 (December 1981).
33. R. A. Finkel, "Issues for distributed operating systems," *Proceedings of INFOCON 82,* 204–205 (1982).
34. Whether a program is a process or merely a portion of a process depends on what is considered to be the machine that provides instruction interpretation. For example, CICS/VS is a single process (task) as far as MVS is concerned. But CICS application programs are processes (tasks) as far as CICS is concerned. CICS is a subtasking monitor, and its applications are interpreted by CICS alone as far as they are concerned.
35. Because the bus transfers that are used to access a common memory can be thought of as small messages, it is possible to build a distributed shared main memory and make it transparent to application programs. To be useful, however, a memory bus has to have very large bandwidths and low delays. This effectively eliminates that approach from consideration as a general-purpose method of interprogram communication.
36. *Customer Information Control System/Virtual Storage (CICS/VS), Version 1 Release 6: General Information,* GC33-0155, IBM Corporation; available through IBM branch offices. Using the various LU 6 levels, CICS supports transparent remote access to files, data bases, queues, and transaction scheduling on other CICS systems.
37. *Distributed Office Support/370 Version 2,* GH12-5139, IBM Corporation; available through IBM branch offices. DISOSS supports distributed access to a document library, formatting, printing, and distribution services.
38. *Network Job Entry for JES2,* GC23-0100, IBM Corporation; available through IBM branch offices. NJE provides network job transmission, output routing, and file transfer services.
39. *File Transfer Program,* GH12-5129, IBM Corporation; available through IBM branch offices.
40. *Distributed Systems Executive Version 2,* GH19-6229, IBM Corporation; available through IBM branch offices. DSX provides centralized management and control of the file contents of multiple distributed processors.
41. *IMS/VS General Information Manual,* GH20-1260, IBM Corporation; available through IBM branch offices. IMS/DC provides message queuing, delivery, routing, transaction scheduling, and formatting services for centralized and distributed configurations.
42. *System/38 Data Communication Programmers Guide,* SC21-7825, IBM Corporation; available through IBM branch offices. System/38 provides transaction scheduling and formatting services for centralized and decentralized configurations of System/38 and CICS via LU 6.2.
43. *System/34 Interactive Communication Feature Reference Manual,* SC21-7751, IBM Corporation; available through IBM branch offices. System/34 ICF provides transparent transaction initiation between System/34 and System/36 via LU 6.0. It also provides similar support via other LU types to IMS and CICS.
44. *System/36 Interactive Communication Feature: Reference,* SC21-7910, IBM Corporation; and *Guide and Examples,* SC21-7911, IBM Corporation; available through IBM branch offices. System/36 ICF provides transparent transaction initiation between System/34 and System/36 via LU 6.0. It also provides similar support via other LU types to IMS and CICS.
45. *Distributed Disk File Facility,* SC21-7869, IBM Corporation; available through IBM branch offices. DDFF provides transparent file access for files contained on System/34 and System/36.
46. Many other products provide distributed services for use in SNA. Consult your IBM representative for additional information.
47. Details of the physical network are not of interest to the LUs, so long as arbitrary session connectivity is supported. SNA architecture layers are defined to supply such session connectivity to LUs. In SNA products available at the time this article was written, arbitrary connectivity was supported for all pairs of LUs, except when the physical topology would cause the sessions to flow through a boundary function, as would be the case in NCP/VS.
48. The architected boundary between an LU and the path control network[3] is not at exactly the same place as the VTAM API used by CICS. Thus, strictly speaking, CICS and part of VTAM correspond to an SNA LU. However, the LU function that is in VTAM (transmission control and parts of LU network services) is a small portion of the total LU. Therefore, it is correct to think of CICS as the LU and VTAM as the transport (or transmission subsystem) component.
49. Some product-specific details are exposed across the network when remote management of network nodes is desired. For example, the load module for a CICS COBOL program does not run on a System/38. In SNA, these kinds of differences are handled by applications, such as DSX,[40] that use the network just like other applications.
50. Some distributed SNA services are provided by programs that run on the SSCP or PU NAUs. For example, the translation of LU name to a network address that is performed during session initiation is a distributed service running on SSCP(s).
51. The canonical (or architectural) description given here does not necessarily correspond to implementation details. For example, that which the architecture describes as several programs running on several LUs might be implemented as

a single monolithic program. This is commonly done to a greater or lesser degree in such SNA terminal products as the IBM 3274 or IBM 5251. In products that support application programs, the SNA notion of an LU is implemented as a portion (or subsystem) of a native operating system. The SNA limb is grafted onto a native trunk when viewed from the perspective of the local operating system, and the local limb is grafted onto the SNA trunk when viewed from the perspective of the network.

52. The first SNA products were the IBM 3600 banking system, NCP release 3, and VTAM release 1. These were shipped in 1974.
53. The bracket protocol used for conversations is defined in Reference 3.
54. The contention winner status is on a per-session basis. When parallel sessions exist between two LUs, each LU is generally the contention winner on some number of the sessions, thereby permitting that LU always to be guaranteed access to some number of sessions, even though the partner LU also wants to use the session. The bidding request used by the contention loser LU is encoded as bits in the request header: BBI set to B '1' either on FM data or on the LUSTAT DFC command. The BID command used with other LU types is not used with LU 6.2.
55. An LU can support more than one LU session type. CICS/VS, for example, supports several LU 0 protocols as well as LU 1, LU 2, LU 3, LU 4, LU 6.1, and LU 6.2.
56. Through use, some LU 0 protocols have become de facto architecture. The LU 0 protocol into which VTAM maps non-SNA IBM 3270 terminals is an example.[57]
57. *ACF/VTAM Version 2 Programming*, SC27-0611, IBM Corporation; available through IBM branch offices.
58. *An Introduction to the IBM 3270 Information Display System*, GA27-2739, IBM Corporation; available through IBM branch offices.
59. At the time this paper was being written, support for LU 6.2 had been announced by CICS/VS, System/38, SCAN-MASTER, and DISPLAYWRITER.
60. W. J. Doherty and R. P. Kelisky, "Managing VM/CMS systems for user effectiveness," *IBM Systems Journal* **18,** No. 1, 143–163 (1979). Pages 154–155 of this paper discuss the improved productivity that results from faster response times.
61. The rationale for parallel sessions is given in detail in Reference 9.
62. The canonical implementation is also referred to as a meta-implementation. Although the SNA meta-implementation[3,10,11] has many of the properties of a real implementation, including the ability to be executed, it omits many features needed by actual implementations. Such features include interfaces to actual hardware (to attach real communication lines), to real operators (no library of operator screens is included), and to interfaces to real programs (only the FAPL language is supported). Cycle usage and storage occupancy are not given the same attention that they receive in products. It is the omissions that make the model node useful. The resulting small size of the model serves to highlight the SNA node-to-node protocols.
63. Products are not required to use the syntax defined in Reference 1. They are required to provide compatible semantics. That is, there must be a mapping from the architected functions to the product-supplied functions. For example, the ALLOCATE verb is implemented using two statements in both CICS and System/38. CICS uses a combination of ALLOCATE and CONNECT PROCESS, whereas System/38 uses the combination of OPEN and EVOKE. Since the function provided by the two combinations matches that of the architected ALLOCATE, both products satisfy the architecture.
64. Other conversation verbs that are not discussed in the text are defined in Reference 1. The omitted verbs deal with details of the model or with LU 6.2 functions that are of lesser importance.
65. Messages are encoded as chains in the session flows. Thus the negative response created by SEND_ERROR causes the usual SNA action of purging to end of chain.
66. Products may provide a level of indirection for some parameters. For example, CICS/VS provides SYSID, which maps indirectly to the target LU_NAME.
67. The MODENAME supplied by the transaction program determines the session level characteristics, such as pacing count, maximum RU size, and session cryptographic support. MODENAME also determines the COSNAME, which determines the class of service to be supplied to the session by SNA's path control.[9]
68. It would have been possible to make the target of the conversation an already-running process or task. This was not done because it would have required that a process-id be created and distributed to users in need of its value. The resulting overhead would not have been compensated by any increase in functional capability. Further, the new-process model is more lenient for implementation because it is easy to simulate a new process with an existing one, but not conversely.
69. R. Obermarck, "Distributed deadlock detection algorithm," *ACM Transactions on Database Systems* **7,** No. 2, 187–208 (June 1982).
70. LU 6.2 defines optional verbs to allow a transaction program to wait for the completion of one of a number of RECEIVEs or other events.
71. SNA's transport and session protocols taken together can be implemented very efficiently. Path lengths between 1 and 2 instructions per byte for sending or receiving can be achieved on small and large machines with packet sizes in the range of 256 to 1024 bytes and messages sizes of 1024 bytes and greater. One reason this efficiency is possible is that, because path control does not discard packets to achieve congestion control, data link control's error recovery brings the failure rate of sessions down to a low value. End-to-end error recovery can then be performed by transaction program algorithms that are needed anyway to handle node failures or device recovery. For example, a "resume printing at page n" command might be needed to handle paper outages. The same command can also be used for session outages if they are infrequent. This contrasts with some other network designs that require end-to-end packet retransmission to recover from relatively frequent packet losses that are created by their own routing and flow control algorithms.
72. B. C. Housel and C. J. Scopinich, "SNA Distribution Services," *IBM Systems Journal* **22,** No. 4, 319–343 (1983, this issue).
73. Compare the SBI command used in LU 6.1. Because the LU 6.1 protocol for termination of sessions is localized to each session, an UNBIND is often followed by a new BIND from the other LU as an attempted error recovery. Only when this fails is the session termination complete.
74. The size referred to is the maximum RU size established by the BIND exchange.
75. LU 6.0 was shipped in CICS/VS 1.4 in 1978. LU 6.1 was shipped in CICS/VS 1.5 in 1980 and in IMS/VS 1.1.6 in 1981. A version of LU 6.0, available in ICF for use between

System/34s since 1980, is now available on System/36. LU 6.1 is defined in Reference 4 and the System/34 version is defined in Reference 43.

Reprint Order No. G321-5197.

James P. Gray *IBM Communication Products Division, P. O. Box 12275, Research Triangle Park, North Carolina 27709.* Dr. Gray joined IBM in 1970 as a research staff member in a research group in Raleigh, North Carolina. He worked on processor architecture in 1970 and 1971, then on network architecture. He continued this work in the Communication Systems Architecture Department after 1972. Dr. Gray has contributed to the definition of several portions of SNA, most recently as a member of the group responsible for defining LU 6.2. He has been the technical assistant to the chairman of the SNA architectural maintenance board since its inception in 1973. Dr. Gray earned a B.E. in electrical engineering from Yale College, New Haven, Connecticut, in 1965 and a Ph.D. in communication theory from the Yale Department of Engineering and Applied Science in 1970.

Peter J. Hansen *IBM Information Systems and Communication Group, 44 South Broadway, White Plains, New York 10601.* Mr. Hansen joined IBM in 1969 in Madison, Wisconsin, as a systems engineer. Since then, he has held technical and managerial positions in several data base and communication-oriented advanced technology projects. In addition to participating in the early design efforts for the IBM 8100 and System/38, he was a member of the group responsible for the definition of LU 6.2. He is currently the manager of processor architecture. Mr. Hansen received his B.S. and M.S. degrees in electrical engineering from the University of Wisconsin, Madison.

Pete Homan *Tandem Computers Incorporated, 19333 Vallco Parkway, Cupertino, California 95014.* Mr. Homan joined IBM at the IBM United Kingdom Laboratories Limited, Winchester, Hampshire, England, in 1970. He worked initially on the PL/I Language and the Virtual Telecommunications Access Method (VTAM). From 1974 to 1979 he was in the development group responsible for the CICS transaction processing system, working on the design and implementation of terminal support and distributed transaction processing function in SNA networks. From 1979 to 1981, he worked on the staff of the Director of Communications Programming and was a member of the group responsible for the definition of LU 6.2. He is currently pursuing an interest in fault-tolerant transaction processing systems. Mr. Homan received a B.Sc. in chemistry in 1969 and an M.Sc. in computer science in 1970 from the University of Birmingham, England.

Michael A. Lerner *IBM Communication Products Division, P. O. Box 12275, Research Triangle Park, North Carolina 27709.* Mr. Lerner is currently a senior engineer manager responsible for advanced network architecture. He joined IBM in 1967 as a systems engineer and in that position was involved in installing both the Airline Control Program and a CICS system. In 1973, Mr. Lerner moved to the Systems Development Division in Kingston, New York, where he became involved in the development of SNA, both in the product development area (IBM 3790) and as an architect in the Communication Systems Architecture organization. He was a member of the group responsible for defining LU 6.2. Mr. Lerner received a B.S. in pharmacy from Columbia University, New York City, in 1963.

Mark Pozefsky *IBM Communication Products Division, P. O. Box 12275, Research Triangle Park, North Carolina 27709.* Since joining IBM in 1979, Dr. Pozefsky has worked on Systems Network Architecture (SNA), first on the SNA 4.2 path control enhancements (explicit and virtual routing) and then on the definition of LU 6.2. He received an Sc.B. in applied mathematics from Brown University, Providence, Rhode Island, in 1970 and the Ph.D. in computer science from the University of North Carolina at Chapel Hill in 1977.