A Communications Interface for Systems Application Architecture

JACK P. SANDERS, MEL R. JONES, JOHN E. FETVEDT, AND MARSHA E. FERREE

Abstract—**The Communications Interface is a major enhancement to IBM's Systems Application Architecture™ (SAA).[1] It provides a consistent, easy-to-use interface between distributed applications and underlying network protocols such as IBM's Systems Network Architecture (SNA) logical unit 6.2 (LU 6.2). This paper describes the requirements for program-to-program communication that prompted the creation of the Communications Interface, discusses the considerations that went into the design of its function, and explains the basic concepts of the interface. An example of two programs using the Communications Interface is also included.**

Introduction

SINCE SNA [1]–[4] was first announced in 1974, there has been a steady evolution within SNA of program-to-program communication and the programming interfaces used for such communications. Although architected logical unit (LU) protocols[2] initially emphasized program-to-device communications [5], program-to-program communication was possible from the outset.[3] SNA's expansion to allow interconnection of multiple host systems in 1977, and the continuing increase in processing power of distributed systems and intelligent workstations, has emphasized the requirement for a complete program-to-program communication architecture.

The first step in IBM's support for such communication was the creation of the LU 6.2 architecture, which defines both the interprogram communication services and the formats and protocols required for general purpose program-to-program communication. The second step was the specification of the Communications Interface, an element of SAA's Common Programming Interface (CPI), which provides a consistent and easy-to-use programming interface to the underlying LU 6.2 program-to-program services.

LU 6.2

A derivative of the LU 6.0 and LU 6.1 host–subsystem architectures, the LU 6.2 architecture is described in a two-part fashion. First, there are the communication services available to programs (collectively, the service interface), which are specified in [6]. The semantics of the service interface are defined by programming-like statements referred to as *verbs*. The second portion of the architecture is the underlying formats and protocols used to support the service interface; these formats and protocols are described in [7]. See also [8] and [9].

The LU 6.2 verbs provide a level of abstraction that reveals only the services available through the programming interface, not the underlying protocols. Systems implementing the LU 6.2 services—or, more generally, any program-to-program communications service—must provide access to those services through a user-friendly application programming interface (API). In the distributed system environment, this API must address the following requirements.

- The syntax of the API should be consistent from system to system in order to ease development of applications that span systems and to reduce the cost of portable applications. For instance, the semantic function of a call to allocate communications resources should have the same syntax across all systems.
- Application programmers using the API should not be required to provide information about the network in order for the system to perform its functions. For example, network-specific values such as remote system names, class-of-service names, and security parameters should not be required. Such information is typically available to the systems programmer rather than to the application programmer.
- There should be a well-defined set of supported high-level languages for the API, thus contributing to ease-of-use and program portability. Support of these languages should not vary from system to system.

Communications Interface

The Communications Interface [10] of IBM's Systems Application Architecture (SAA) was created in order to provide a consistent interface for communications across the SAA environments, while, at the same time, establishing an interface to LU 6.2 services that would address the requirements described above. Although the Communications Interface is initially provided on top of LU 6.2 network services, other types of underlying network support are possible, so long as that network support is

Manuscript received June 10, 1988; revised March 2, 1989.

The authors are with IBM Corporation, Research Triangle Park, NC 27709.

IEEE Log Number 8929691.

[1]Systems Application Architecture is a trademark of International Business Machines Corporation.

[2]Within SNA, a *logical unit* enables end users such as persons, devices, or programs to gain access to network resources.

[3]LU type 0 allows use of unarchitected, private protocols between programs.

0733-8716/89/0900-1073$01.00 © 1989 IEEE

sufficiently rich.[4] The new interface specifies a syntax that is the same on all SAA systems and is independent of the underlying network support; it eliminates the requirement for the application programmer to specify network-specific information; and it is provided across a uniform set of languages on each implementing system.

The following sections provide a brief overview of SAA and the role of the Communications Interface within it, and then discuss the design considerations and constraints for the interface more thoroughly. A tutorial on the basic concepts of the interface is provided and, finally, an example program flow shows how two programs communicate using the interface.

SAA Overview

IBM announced SAA [11], in March 1987, as a collection of software interfaces, conventions, and protocols designed to increase the consistency and connectivity of the participating systems.[5] SAA has four primary objectives:

- improve customer productivity, through transferable programming skills
- allow easier porting of program code
- provide simpler and more uniform user access
- simplify the development of applications that span systems.

To meet these objectives, SAA addresses four major areas of function:

- *Common User Access*—the design and use of menus, screen panels, graphics, and other user-interaction techniques.
- *Common Programming Interface* (CPI)—the languages and services that application programmers use in developing software. The list of SAA languages includes Cobol, Fortran, C, RPG, Procedures Language, and Application Generator. The services include communications, database, dialog, presentation, and query.
- *Common Communications Support* (CCS)—the protocols for interconnection of systems and programs.
- *Common Applications*—the applications built by IBM and other software vendors.

LU 6.2 is an element of the CCS—it defines the communication protocols necessary for the support of interconnected programs—while the Communications Interface is part of the CPI.

[4]For example, the facilities of the ISO Distributed Transaction Processing Draft Proposal are capable of supporting the Communications Interface.

[5]SAA defines a number of operating environments for which the various SAA elements will be implemented:

- TSO/E in the Enterprise Systems Architecture/370™
- CMS in the VM/System Product or VM/Extended Architecture
- Operating System/400™(OS/400™)
- Operating System/2™(OS/2™) Extended Edition
- IMS/VS Data Communications in the Enterprise Systems Architecture/370™
- CICS/MVS in the Enterprise Systems Architecture/370™.

Operating System/2, Operating System/400, Enterprise Systems Architecture/370, OS/2, and OS/400 are trademarks of the International Business Machines Corporation.

Communications Interface—Design Objectives

Because the Communications Interface is defined within the context of SAA—specifically as an element of the CPI, but also by its use of the LU 6.2 protocols within CCS—the design objectives of the Communications Interface are, in part, derivable from the broader SAA objectives of consistency and connectivity.

The following sections discuss the major design points of the interface in greater detail.

Program-to-Progam Communication Support

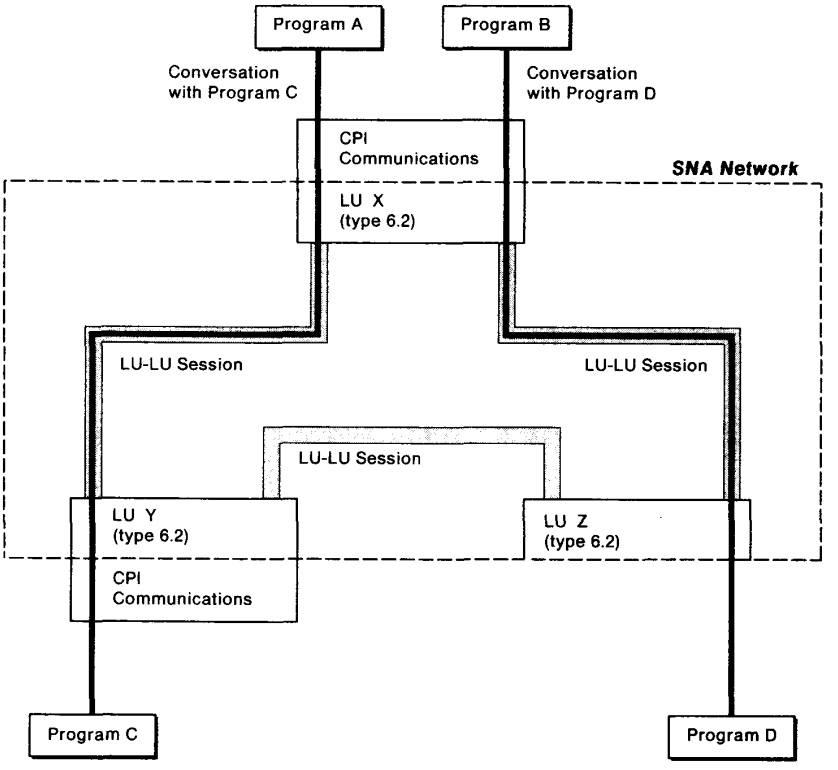
As that part of the CPI providing the communication service, the Communications Interface was required to support general-purpose program-to-program communication for a wide range of applications using communications and distributed processing. At the same time, the specification of LU 6.2 as the session services element of the CCS meant that the Communications Interface would build on the underlying protocol support of LU 6.2, thus making it desirable that the interface correspond closely to LU 6.2. These factors led to the adoption of the LU 6.2 *conversation* (described below) as the interface's underlying model of operation. Viewed from this perspective, the Communications Interface complements the semantic definition of the LU 6.2 service interface and provides a consistent syntactic definition across all SAA systems.

Conversations

A *conversation* is a logical connection between two programs and is the basic construct within which communication between the two programs takes place. Programs can start conversations with partner programs and, on those conversations, perform such operations as sending and receiving data, synchronization of processing, and error notification. Communication on a conversation is half-duplex; the right to send is exchanged under program control. A program may have several conversations active simultaneously.

Fig. 1 illustrates a configuration of progams using the Communications Interface to exchange data and information through an SNA network. The conversations the programs use to communicate are shown as single bold lines between the programs. The doubly-lined and shaded areas are LU 6.2 *sessions*. Sessions are the logical connections between LU's upon which conversations are transported. The two communicating programs have exclusive use of a session for the duration of their conversation; when the conversation is ended, the session is available for use by another conversation.

A consequence of the design decision to use LU 6.2 conversations and functional content is that Communications Interface programs can communicate with programs written to existing LU 6.2 API's. In Fig. 1, Program *B* is using the Communications Interface for its conversation with Program *D*, but Program *D* is using an existing LU



Note:
Programs *A*, *B*, and *C* use the Communications Interface. Program *D* uses an existing LU 6.2 application programming interface (API).

Fig. 1. Application programs using the Communications Interface to converse through an SNA Network.

6.2 API. A large number of systems currently provide a programming interface to LU 6.2.[6]

Interface Consistency

As part of SAA's CPI, the Communications Interface is required to be consistent across systems in each of several aspects: concepts, semantics, functional content, and syntax. The conversation is the basic concept within which the functions of the interface are provided. Its characteristics have already been described. Consistency of syntax and semantics ensures that a program using the interface has identical results regardless of the system on which it executes; this is a fundamental requirement for program portability. Consistency of functional content assures that the same communications services are available on each system.

Besides simplifying the porting of programs, interface consistency has advantages for programmers building applications that span systems. By the provision of a single syntax, semantics, and set of concepts, the Communications Interface allows programming skills for communications to be applicable across systems. Similarly, testing across multiple environments is simplified.

An additional requirement for CPI Communications was that the interface be as consistent as possible across the different SAA languages in order to enable the programmer to move easily between programming languages. The language construct chosen was that of a subroutine call, a position-dependent specification of call name and parameter list that was available in all of the SAA languages. By using an existing language construct, no new language extensions were required.

The decision to use a subroutine call means that the only differences across systems in the call syntax will be those minor syntactic differences that result from language differences. This allows the application programmer who has become familiar with the Communications Interface in one language to use it from any other SAA language without an additional learning requirement beyond that of knowing the language.

[6]IBM systems that provide an LU 6.2 API include CICS, System 38, System/36, Series/1, APPC/PC, System/88, RT PC, VTAM (with application), OS/2 Extended Edition, VM/SP, and OS/400.

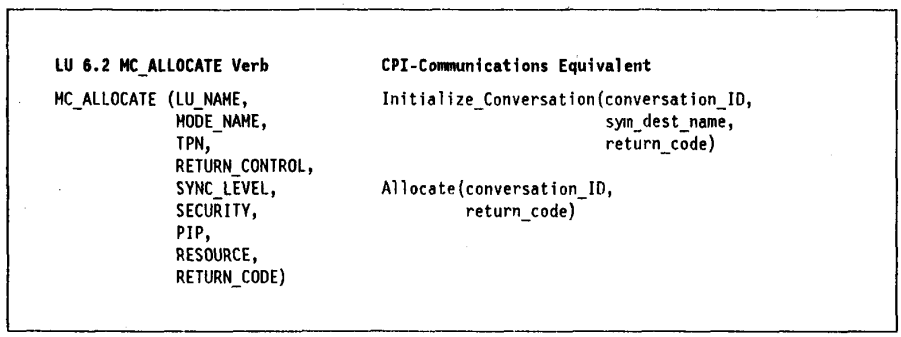


Fig. 2. LU 6.2 MC_ALLOCATE versus CPI-Communications Equivalent.

Reduced Complexity for the Application Programmer

A design premise of the Communications Interface was that programmers of applications that need only simple communication services should not be required to have a detailed understanding of communications. A number of techniques were used to reach this end—these are discussed in greater detail in the "Basic Concepts" section that follows—but the underlying principle was that the interface be simplified while continuing to provide a complete set of communications services.

Communications Interface—Basic Concepts

The Communications Interface reduces the knowledge required for communications programming by defining a set of *conversation characteristics*. Default values for the characteristics are established for each conversation, thereby shielding programmers from the need to understand function that is not required for a specific application.

By way of example, Fig. 2 shows a comparison of the LU 6.2 MC_ALLOCATE verb [6] with the analogous Communications Interface calls that are used to provide the same function. A program issues MC_ALLOCATE to request the LU to allocate the communications resources necessary for a conversation with a specified partner program. Within the Communications Interface, the Initialize_Conversation call is used to establish the default conversation characteristics; the Allocate call then performs the same function that the MC_ALLOCATE does.

In comparison to the nine parameters specified for the LU 6.2 verb interface, only three parameters are required for the Communications Interface. Two of these, *conversation_ID* and *return_code*, require no special communications knowledge: *conversation_ID* is used to identify a conversation (it is provided by the Communications Interface on the Initialize_Conversation call, then reused on the Allocate), while *return_code* simply specifies the result of the call execution.

The last parameter, *sym_dest_name*, represents a *symbolic destination name*, which, in turn, specifies the destination information required by the LU to allocate the conversation. This information is maintained by the system administrator in a system entity known as *side information*.

Side Information

The side information is indexed by the *sym_dest_name* and contains, for each entry, the following three pieces of information:[7]

- *partner_LU_name*—indicates the name of the LU where the partner program is located.
- *mode_name*—designates properties of the session to be allocated for the conversation. Mode_names are defined by the network owner, apply to sessions, and may be, for example, SECURE, BATCH, or FAST, corresponding to properties of the session and its underlying transport facilities.
- *TP_name*—specifies the name of the remote program.

Returning to the example of Fig. 2, the LU_NAME, MODE_NAME, and TPN parameters are reproduced by the Communications Interface via the side information indexed by the *sym_dest_name*. For advanced communications programming, this destination information may be modified after the Initialize_Conversation call by specific Communications Interface calls (Set_Partner_LU_Name, Set_Mode_Name, and Set_TP_Name). The remaining function to be duplicated (represented by the RETURN_CONTROL and SYNC_LEVEL parameters[8]) is provided in the same fashion (via the Set_Return_Control

[7]As described here, symbolic destination name specifies destination information for a partner program within an SNA network. In a different communications environment, symbolic destination name could specify destination information appropriate to that environment.

[8]SECURITY parameters for authorization and access control in the system where the partner program is located are specified in ways other than through the Communications Interface, and PIP (program initialization parameters) is not currently supported; RESOURCE and RETURN_CODE correspond to *conversation_ID* and *return_code*, respectively.

and Set_Sync_Level calls, respectively). This approach allows programs to use the default set of conversation characteristic values or, for advanced function, to exercise more detailed control over the conversation via separate calls. A more complete discussion of these calls is provided in the next section.

Program Calls

The principle of "easy" versus "advanced" was used throughout the interface, and results in a natural dichotomy of function between what the Communications Interface refers to as *starter-set* and *advanced-function* calls:

- *Starter-Set Calls:* Starter-set calls allow for simple communication of data between two programs and assume the program uses the default values for the Communications Interface conversation characteristics. No other calls are required beyond this set for simple program-to-program communication.
- *Advanced-Function Calls:* Advanced-function calls allow complete control over the characteristics of the conversation and specific selection (via the Set calls) of the level of function provided.

Table I lists the two groups of Communications Interface calls. Of the starter-set calls, Initialize_Conversation and Allocate have already been shown in allocating a conversation. Their counterpart, for the program at the other end of the conversation, is the Accept_Conversation call, which establishes initial conversation characteristics for the partner program (each of the programs sharing a conversation has its own local view of that conversation) and returns a *conversation_ID* to that program for use on succeeding calls. Send_Data and Receive are used to exchange data between programs, and Deallocate allows a program to end a conversation.

The advanced-function calls can be grouped by function. As already shown, the Set calls allow a program to modify conversation characteristics, while Extract calls allow a program to examine the values of certain conversation characteristics. The Set calls also prevent programs from making invalid changes to conversation characteristics. For example, if a program attempts to change the *conversation_type* after the conversation has already been established with Allocate (an illegal change), the Communications Interface informs the program of its error and disallows the change.

Confirm and Confirmed provide an explicit means of program synchronization. A program wishing to have an acknowledgment from its partner program that data have been satisfactorily received (that is, received and processed to an agreed-to-level) issues Confirm; the partner program replies with Confirmed to indicate the satisfactory receipt of the data.

Prepare_To_Receive, Request_To_Send, and Test_Request_To_Send_Received are all required because of the half-duplex nature of the conversation. They allow additional program control over the direction of the data flow. Prepare_To_Receive is used to transfer the right to send, along with any data that may still be retained by the LU, to the partner program. A program that is receiving data and wishes to begin sending data issues Request_To_Send, and Test_Request_To_Send_Received is used by a sending program to check whether such a request has been received.

The Flush and Send_Error calls allow, respectively, a program to request that data be sent immediately or to notify the partner program that an error has occurred.

TABLE I
BREAKDOWN OF CALLS BETWEEN STARTER SET AND ADVANCED FUNCTION

Starter Set	
Initialize_Conversation	
Accept_Conversation	
Allocate	
Send_Data	
Receive	
Deallocate	
Advanced Function	
Confirm	Set_Conversation_Type
Confirmed	Set_Deallocate_Type
Flush	Set_Error_Direction
Prepare_To_Receive	Set_Fill
Request_To_Send	Set_Log_Data
Send_Error	Set_Mode_Name
Test_Request_To_Send_Received	Set_Partner_LU_Name
	Set_Prepare_To_Receive_Type
	Set_Receive_Type
Extract_Conversation_Type	Set_Return_Control
Extract_Mode_Name	Set_Send_Type
Extract_Partner_LU_Name	Set_Sync_Level
Extract_Sync_Level	Set_TP_Name

Conversation Characteristics

As discussed already, the Communications Interface maintains a set of characteristics for each conversation. These characteristics are established for each program on a conversation basis, and the initial values assigned to the characteristics are determined based on the program's role in starting the conversation (that is, whether the program issued Initialize_Conversation or Accept_Conversation).

Table II provides a comparison of the conversation characteristics and initial values as set by the Initialize_Conversation call and the Accept_Conversation call. The symbolic values (prefixed by "CM_") shown in the table are pseudonyms that represent integer values. Conversation characteristics are used to modify the action of the calls. For example, *receive_type* specifies whether a receive operation will wait for information or complete immediately if none is available.

The conversation characteristics used in the side information have already been described; two other characteristics apply to the conversation as a whole and therefore may not be modified after the conversation has been started with the Allocate call:

- *conversation_type* indicates whether the conversation is *basic* or *mapped*. Mapped conversations allow pro-

TABLE II
CHARACTERISTICS AND THEIR DEFAULT VALUES

Name of Characteristic	Initialize_Conversation sets it to:	Accept_Conversation sets it to:
conversation_type	CM_MAPPED_CONVERSATION	The value received on the conversation start-up request
deallocate_type	CM_DEALLOCATE_SYNC_LEVEL	CM_DEALLOCATE_SYNC_LEVEL
error_direction	CM_RECEIVE_ERROR	CM_RECEIVE_ERROR
fill	CM_FILL_LL	CM_FILL_LL
log_data	Null	Null
log_data_length	0	0
mode_name	The mode name from side information referenced by *sym_dest_name*	The mode name for the session on which the conversation start-up request arrived
mode_name_length	The length of *mode_name*	The length of *mode_name*
partner_LU_name	The partner LU name from side information referenced by *sym_dest_name*	The partner LU name for the session on which the conversation start-up request arrived
partner_LU_name_length	The length of *partner_LU_name*	The length of *partner_LU_name*
prepare_to_receive_type	CM_PREP_TO_RECEIVE_SYNC_LEVEL	CM_PREP_TO_RECEIVE_SYNC_LEVEL
receive_type	CM_RECEIVE_AND_WAIT	CM_RECEIVE_AND_WAIT
return_control	CM_WHEN_SESSION_ALLOCATED	Null
send_type	CM_BUFFER_DATA	CM_BUFFER_DATA
sync_level	CM_NONE	The value received on the conversation start-up request
TP_name	The program name from side information referenced by *sym_dest_name*	Null
TP_name_length	The length of *TP_name*	0

grams to exchange arbitrary data records in data formats agreed upon by the application programmers. Basic conversations indicate that the data exchanged will adhere to a standardized format. This format is a stream of data containing 2-byte length fields that specify the amount of data to follow. Each grouping of "length field, data" is referred to as a *logical record.*

- *sync_level* determines whether any synchronization of data transmission (via Confirm and Confirmed) between the partner programs will be performed on the conversation.

The remaining characteristics modify the action of particular calls:

- *deallocate_type* determines how the conversation will be deallocated, specifically, whether Deallocate will transmit data before finishing the conversation, or transmit data and wait for a confirmation before finishing the conversation.
- *error_direction* defines the type of error being reported by Send_Error, that is, whether the error occurred in data received, or while gathering data to send a reply.
- *fill* determines, for a basic conversation, whether the program receives data as logical records or receives data independent of the logical record format.
- *log_data* and *log_data_length* are used, for a basic conversation, to indicate error information that is placed into system error logs of the partner LU's.
- *prepare_to_receive_type* determines whether the Prepare_To_Receive call will include the function of Confirm or Flush.
- *receive_type* determines whether a Receive operation will wait for information if none is immediately available.
- *return_control* determines whether a program waits for a session or, if a session is not available, receives control immediately after an Allocate operation.
- *send_type* controls how the data will be transmitted. Specifically, it determines whether Send_Data will allow data to be buffered before transmission; cause data to be transmitted at once; prompt the partner program to confirm satisfactory receipt of the data once the data are received; send the data and allow the partner program to begin sending; or send the data and finish the conversation.

For simple communications programming, a program may use the initial set of conversation characteristics provided by the Communications Interface as defaults. A program wishing greater control over the conversation characteristics can issue one of the Set or Extract calls.

PROGRAM FLOW—STATES AND TRANSITIONS

LU 6.2 makes use of conversation states to describe the status of a conversation, and the Communications Interface continues this use. A Communications Interface conversation can be in one of the following states:

Conversation State	Description
Reset	There is no conversation for this *conversation_ID*.
Initialize	Initialize_Conversation has completed successfully and a *conversation_ID* has been assigned for this conversation.
Send	The program is able to send data on this conversation.
Receive	The program is able to receive data on this conversation.
Send-Pending	The program has received both data and permission to send on the same Receive call.
Confirm	A confirmation request has been received on this conversation.
Confirm-Send	A confirmation request and permission to send have both been received on this conversation. After responding with Confirmed, the local program enters **Send** state.
Confirm-Deallocate	A confirmation request and deallocation notification have both been received on this conversation. After responding with Confirmed, the conversation is deallocated.

A conversation starts out in **Reset** state and progresses through the different states listed, depending on the calls made by the program for that conversation and the information received from the remote program. The current state of a conversation determines what calls the program can or cannot make. For example, a program in **Receive** state cannot issue a Send_Data call.

COMMUNICATIONS INTERFACE—EXAMPLE PROGRAM FLOW

This section provides an example program flow between the Communications Interface programs. The call parameter lists shown in the flow are not syntactically complete or precise in order to simplify their presentation; only the parameters of interest are shown. The steps shown in Fig. 3 are described below:

Step	Description
1	Program *A* uses the Initialize_Conversation call to tell the Communications Interface that it wants a conversation with a partner program. The *sym_dest_name* identifies the necessary destination information.
2	No errors are found on the Initialize_Conversation call, and the *return_code* is returned as CM_OK. The Communications Interface also returns a unique *conversation_ID* to Program *A*. The program stores the *conversation_ID* and uses it on all subsequent calls intended for that conversation. At this point, the Communications Interface has established a default set of conversation characteristics for the conversation, based on the *sym_dest_name*, and uniquely associated them with the *conversation_ID*.
3	Program *A* asks that a conversation be started with an Allocate call using the *conversation_ID* previously assigned by the Initialize_Conversation call. If a session between the LU's is not already available, one is activated. Program *A* and Program *C* can now have a conversation.
4	A *return_code* of CM_OK indicates that the Allocate call was successful and the LU has allocated the necessary resources to the program for its conversation. Program *A*'s end of the conversation is now in **Send** state and Program *A* can begin to send data.
5 and 6	Program *A* sends data with the Send_Data call and receives a *return_code* of CM_OK.
7–10	Program *A* makes use of an advanced-function call, Prepare_To_Receive, which will send an indication to Program *C* that Program *A* is ready to receive data. Because of the way the LU buffers data, the conversation startup request may not have been sent yet. This call flushes the data buffer, causes the conversation to be started for Program *C*, and places Program *A* into **Receive** state.
11–13	Program *C*, started by System *Y*'s reception of the conversation startup request and buffered data, makes the Accept_Conversation and Receive calls. Program *A* finishes its processing and issues its own Receive call. It will now wait until data are received (Step 15).
14–16	The *status_received* on the Receive call made by Program *C*, which is set to CM_SEND_RECEIVED, tells Program *C* that it can now issue the Send_Data call. Program *A* receives the data.

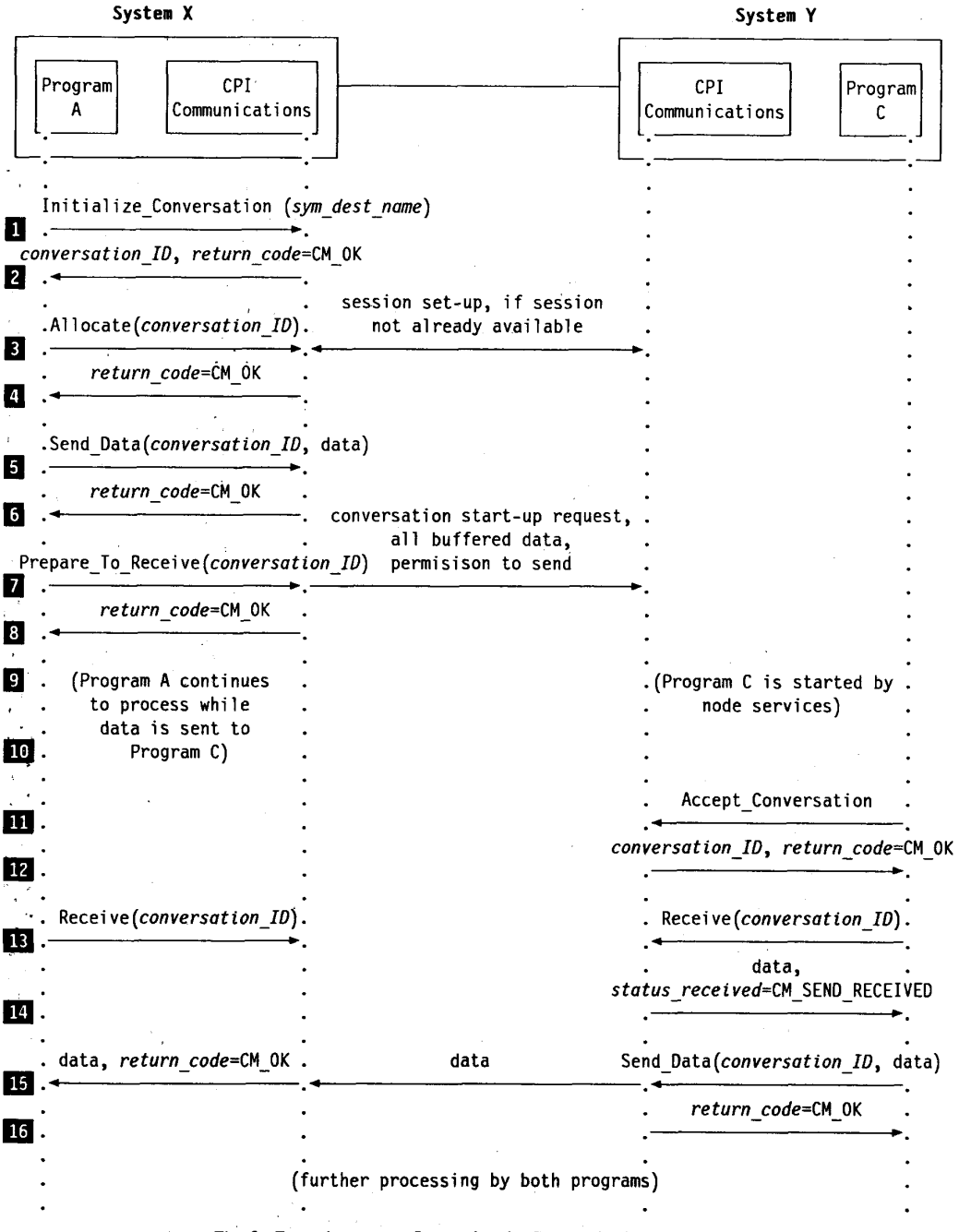


Fig. 3. Example program flows using the Communications Interface.

Concluding Remarks

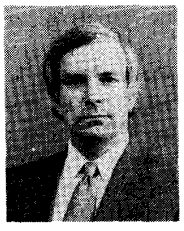
The requirements for simplified and cross-system consistent application-program access to the communications services of LU 6.2 led to the definition of SAA's Communications Interface. The evolving SAA Common Programming Interface, and the Communications Interface within it, provide a programming environment of languages and services for development of application programs that span systems. With the initial implementation of CMS VM/SP (release 6), the Communications Interface provides a base for application-program development by IBM, software vendors, and computer-network owners.

References

[1] *Systems Network Architecture: Technical Overview*, IBM, IBM Form GC30-3073-2, 1986.

[2] *Systems Network Architecture: Concepts and Products*, IBM, IBM Form GC30-3072-3, 1986.

[3] R. J. Sundstrom and G. D. Schultz, "SNA's first six years: 1974–1980," in *Proc. Fifth Int. Conf. Comput. Commun.*, Atlanta, GA, Oct. 27–30, 1980, pp. 578–585.
[4] R. J. Sundstrom, J. B. Staton III, G. D. Schultz, M. L. Hess, G. A. Deaton, Jr., L. J. Cole, and R. M. Amy, "SNA: Current requirements and direction," *IBM Syst. J.*, vol. 26, no. 1, pp. 13–36, 1987.
[5] *Systems Network Architecture: Sessions Between Logical Units*, IBM, IBM Form GC20-1868-2, 1981.
[6] *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, IBM, IBM Form GC30-3084-2, 1985.
[7] *SNA Format and Protocol Reference Manual: Architecture Logic for LU Type 6.2*, IBM, IBM Form SC-30-3269-3, 1985.
[8] J. P. Gray, P. J. Hansen, P. Homan, M. A. Lerner, and M. Pozefsky, "Advanced program-to-program communication in SNA," *IBM Syst. J.*, vol. 22, no. 4, pp. 298–318, 1983.
[9] *An Introduction to Advanced Program-to-Program Communication*, IBM, IBM Form GG24-1584-1, 1986.
[10] *SAA CPI Communications Reference*, IBM, IBM Form SC26-4399-1, 1988.
[11] *SAA Overview*, IBM, IBM Form GC26-4341-2, 1988.

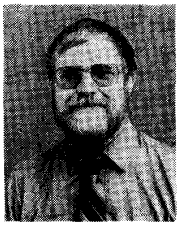


Jack P. Sanders received the Ph.D. degree in mathematics from the University of Virginia and pursued an academic career before joining IBM's Systems Network Architecture group in 1981.

He is a Senior Engineer and manager of the Communication Architecture Strategy Department at IBM's Research Triangle Park facility. Previously he managed the LU 6.2 Architecture department, with responsibility for LU 6.2 and the Communications Interface of the SAA Common Programming Interface.

Mel R. Jones received the B.S. degree in mathematics from Case Western Reserve University, Cleveland, OH, in 1982 and the M.S. degree in computer science from the University of Vermont, Burlington, in 1984.

He has worked for IBM since 1982, and as part of the Systems Network Architecture group at IBM's Research Triangle Park facility since 1985. He has had responsibility for the communications element of the SAA Common Programming Interface (CPI Communications) since 1987.



John E. Fetvedt received the B.A. degree in chemistry from the University of Minnesota, Duluth, in 1965.

He is currently a Senior Programmer at IBM Corporation, Raleigh, NC, and has worked on a wide variety of products, with emphasis on communications and distributed processing since joining IBM in 1965.

Marsha E. Ferree received the B.S. degree in computer science from North Carolina State University in 1979.

She joined IBM's Systems Network Architecture group in the Research Triangle Park in June of that year. She was involved with the definition of the LU 6.2 architecture from its beginning until 1983. She worked on the design of the APPC/VTAM software product from 1983 to 1984. She was responsible for defining the SAA CPI Communications. She recently returned from an assignment with the IMS software product in Santa Teresa, CA. She is now involved in the definition of network standards for computer-to-switch voice/data protocols.