CPI-C: An API for distributed applications

by W. S. Arnette
A. D. Kshemkalyani
W. B. Riley
J. P. Sanders
P. J. Schwaller
J. C. Terrien
J. Q. Walker

Distributed applications have fostered the standardization of application programming interfaces for the underlying communication services. Three popular communication models—remote procedure calls, messaging and queuing, and conversations—support distributed applications across different networking protocols and physical media. Access to the conversational services of Advanced Program-to-Program Communication and Open Systems Interconnection-Distributed Transaction Processing is provided by the Common Programming Interface for Communications (CPI-C), a standard, easy-to-use interface for communication programming. This paper introduces the basic concepts of CPI-C, describes the conversation services available to programs, and presents examples of CPI-C programming.

Reducing the cost of application development had become a major industry focus by the 1980s. Application programmers were clamoring for standard, easy-to-use application programming interfaces (APIs) as a means of improving their productivity and the portability of their programs. System providers, seeing the cost of application development as an inhibitor to their own profitability and growth, realized that standard APIs would be key to inducing independent software vendors to write applications for their systems. Industry groups were promoting standardization of APIs in various environments.

Nowhere was the development of standard APIs more critical than for communication programming, which must deal with the complexities of networks and heterogeneous platforms and systems. In some environments, a *de facto* standard was in place. In Transmission Control Protocol/Internet Protocol (TCP/IP) networks, for example, the sockets interface was a standard API for developing distributed applications. In local area networks, the NetBIOS API was gaining acceptance. Within Systems Network Architecture (SNA) networks, Advanced Program-to-Program Communication (APPC), or LU (logical unit) 6.2, provided support for communication programs, and was widely implemented across both IBM and non-IBM systems. APPC[1-3] defined a rich set of application services for peer-to-peer communication and transaction processing. But it did not define a cross-product API syntax for these services, and a number of product-specific APIs had been developed. The Common Programming Interface for Communications (CPI-C) was designed to provide a standard, easy-to-use API for APPC, and is the topic of this paper.

We begin by briefly discussing the initial objectives for CPI-C and the changing environment that has

Figure 1 CPI-C implementations

IBM CPI-C IMPLEMENTATIONS

AIX SNA Server/6000
APPC Networking Services for Windows
CICS/ESA*
IBM 4680 Store System
IBM DOS/VSE
IBM/ESA*
MVS/ESA*
Networking Services/DOS
OS/2* Communications Manager*
OS/400*
VM/ESA*
VM PWSCS

NON-IBM CPI-C PROVIDERS

Apertus Technologies, Inc.
Apple Computer, Inc.
Attachmate Corp.
Brixton Systems Inc.
Cleo Communications
Compsoft International Inc.
Data Connection Limited
Data General Corp.
HeteroGenius Systems Limited
Hewlett-Packard Co.
Insession Inc.
Microsoft Corp.
NetSoft
Novell Corp., Inc.
SunConnect
Tangram Enterprise Solutions, Inc.
Unisys Corp.
Wall Data Inc.

*Trademark or registered trademark of International Business Machines Corporation.

shaped its development. We then introduce the basic concepts of CPI-C and describe the conversation services available to CPI-C programs. Finally we illustrate the use of CPI-C for distributed applications with CPI-C programming examples using portable C code.

CPI-C evolution

This section traces the evolution of CPI-C and compares the CPI-C model to other communication models.

Initial objectives. Two major requirements affected the initial design of CPI-C.[4] *Ease of use* meant hiding as much as possible of the complexity of communication programming. *Standard access to APPC services* shaped the fundamental communication model and services CPI-C provides. The "standard" aspect of CPI-C depended on the acceptance of CPI-C among system providers and vendors throughout the industry.

CPI-C included ease-of-use features from its initial definition. Application programmers can use the familiar subroutine call model of programming languages with consistently defined language bindings. Programmers can use a local alias for the partner program and are shielded from network-specific values. A *starter set* of calls is defined for simple half-duplex communication, and programmers can ignore functions not required for basic use.

Just as the "sockets" application programming interface to TCP/IP reflects the underlying support of the Internet protocols, so does CPI-C reflect the underlying services and protocol support of APPC: CPI-C supports connection-oriented, peer-to-peer communication that is also well-suited to client/server applications. It allows flexible data exchange and provides a full range of application-level functions, such as program synchronization and error notification. Equally important is support not reflected in the API but provided by the network: flow control, reliable delivery of data, and outage notification.

CPI-C has been implemented in IBM and non-IBM systems. Figure 1 shows a list of most CPI-C implementations or providers that existed at the time this paper was written.

Changing environment. The subsequent development of CPI-C was influenced by advances in technology and industry trends in distributed processing. Application requirements for full-duplex communication, the growing importance of client/server processing, the deployment of distributed network services, and the approval of the Open Systems Interconnection-Distributed Transaction Processing (OSI TP) standard led to CPI-C extensions. The process by which extensions to CPI-C are defined was influenced by the increasing demand for standardization of APIs within industry consortia and formal standards bodies.

The emerging importance of client/server processing led to early CPI-C extensions to address the special requirements of servers. The X/Open** consortium added server support in its 1990 version of CPI-C,[5] with extensions for name registration, for accepting multiple incoming conversations, and for nonblocking calls. In 1992, IBM defined extensions to assist a server to manage the work done on behalf of multiple clients. Additional nonblocking support was added in 1994.

The scope of industry groups now includes the specification of complete programming environments with integrated suites of related services. For example, the Open Software Foundation's Distributed Computing Environment (OSF DCE) includes integrated support for communication, directory, security, and other services. Similarly, as distributed network services such as directory or security pervaded networks, their integration with the communication services provided by CPI-C was required. Support for a distributed directory and a distributed security service was added to CPI-C in 1994.

The OSI TP standard,[6] completed in 1992, defines a set of conversational services similar to those of APPC, but had no API. In its latest version, CPI-C was extended to provide access to the OSI TP services.

Protocol independence has become an important aspect of network applications. Network owners want to be able to choose applications independently of the protocols used within the underlying network. CPI-C can run on either APPC or OSI TP, and programs can be written independently of which of these protocols is being used. Multiprotocol Transport Networking[7] support allows CPI-C programs supported by APPC to use connections provided by either SNA or TCP/IP protocols, transparently to the applications.

Initially developed by IBM in 1988, CPI-C became part of the X/Open Common Applications Environment in 1992.[8] In the formal standards arena, CPI-C was adopted by the OSI Implementers' Workshop as an API for access to OSI TP services. More recently, IBM created the CPI-C Implementers' Workshop (CIW), an open forum of CPI-C implementers and users. The goals for CIW are to extend the CPI-C architecture and to promote its implementation and use. The CIW defined the most recent version, CPI-C 2.0,[9] in 1994, and work continues within the CIW on further extensions. The CIW maintains a liaison with X/Open, which has published CPI-C 2.0 as a preliminary specification.[10]

Comparison to other communication models

Within the framework of IBM's Open Blueprint*,[11] three communication models—*remote procedure calls, messaging and queuing*, and *conversations*—support distributed applications. The Open Blueprint is a guide to distributed computing by providing customers with a structure to organize products and applications in an open distributed environment. These models present a range of alternatives for communication.

To an application program, a remote procedure call (RPC)[12] is like a local subroutine call. The program issues the call, and the called program, which typically executes somewhere else in the network, is invoked. RPC software transfers the input and output parameters of the call between systems. Data exchange follows a simple request/response model. The program is shielded from the complexities of network programming and is not able to react to network problems.

CPI-C supports connection-oriented, peer-to-peer communication suited to client/server applications.

The messaging and queuing (MQ)[13] model is asynchronous. Programs put messages on queues and take messages off queues; communication with another program is indirect through the queues. Data exchange is flexible; a program can enqueue multiple messages at a time, as appropriate to the application, and must correlate responses with requests. A store-and-forward capability allows a message to be held until the program is available. The program is not aware of network outages and does not use network-specific values, such as addresses, which are handled for the program by the underlying queue management software.

A program using CPI-C establishes a logical connection, or *conversation*, to communicate with a partner program. Data exchange can be a simple one-way message, or multiple messages sent and received by both partners. The program can transfer data efficiently over long-lived connections, synchronize processing with the partner program, notify the partner of errors, and react to diagnostic information in the event of failures.

Each of the three communications models has its advocates, and each will continue to play an important role in application development. RPC, with its simple interface, will serve basic request/response processes, especially where a client requester knows its server will return a fixed response, and cares little about detecting and reacting to network problems. Messaging and queuing will appeal to those interested primarily in asynchronous, time-independent processing. The CPI-C conversational model will best serve a wide range of distributed computing needs, running the gamut from synchronous, time-critical applications to sophisticated database applications.

In the following sections, we expand on the CPI-C conversational model and demonstrate, through examples, its ease of use.

Basic CPI-C concepts and conversation services

This section introduces the basic concepts of CPI-C and the conversation services it provides. For the complete specification of CPI-C, see Reference 9. We use simple examples to illustrate how to use CPI-C. Appendix A contains complete source code for the example programs in the C programming language, which today is available on almost all systems.

Call interface. CPI-C provides a subroutine call interface and defines language bindings for a number of programming languages, including C, COBOL, FORTRAN, REXX, PL/I, RPG, and CSP (Cross System Product). Except for minor differences in call syntax, CPI-C has the same appearance across these languages.

Conversations and their characteristics. Since CPI-C is a conversational interface, every CPI-C application consists of at least two programs, one program on each computer where part of the application is to run. Programmers have to design, code, and test the peer programs in tandem. Programs issue calls to CPI-C to establish a conversation, to exchange data and perform other processing on that conversation, and to terminate the conversation when it is no longer needed. A program may have multiple conversations and partners simultaneously.

At the time a conversation is established, each program receives a local *conversation identifier* for the conversation. Each program uses its conversation identifier on all subsequent calls on that conversation.

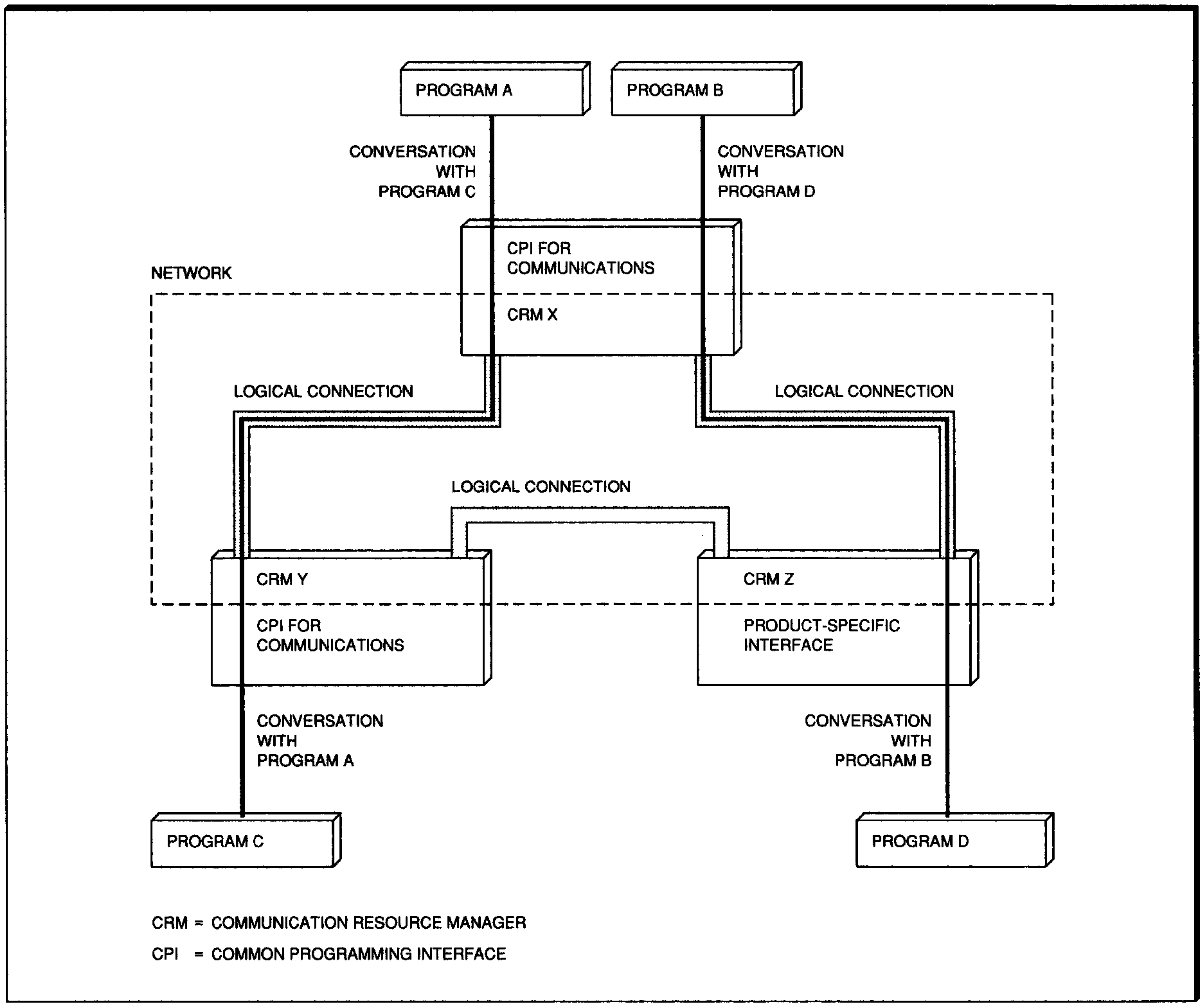
CPI-C maintains a set of *conversation characteristics* at each end of a conversation. Some conversation characteristics contain destination information that is the addressing and security information necessary to establish a conversation. Other characteristics specify a function level for the conversation. An example is the sync_level characteristic that specifies the level of synchronization support (e.g., for a distributed database application) that the programs can use on the conversation. Default values, assigned when the conversation is established, allow for the creation of simpler program logic. A program may view (using Extract calls) and modify (using Set calls) the values of conversation characteristics.

A system administrator can store destination information regarding the partner program in local *side information* to simplify establishing a conversation. The program initiating the conversation identifies the appropriate side information entry for its conversation by a *symbolic destination name* on the Initialize_Conversation call. CPI-C uses the side information entry to assign initial values of the corresponding conversation characteristics.

CPI-C conversations have states that constrain program actions. As the program issues calls to CPI-C, the conversation makes transitions from one state to another. For example, on a half-duplex conversation, when the program in a send state has finished sending data, it can change to a receive state by sending a permission-to-send indicator to the partner. The state of the conversation is local; that is, the states of the conversation as seen by the two program partners may be different at a particular instant. A program may determine the state at its end of the conversation by issuing the Extract_Conversation_State call.

System services. System services such as program startup and termination processing, context management, directory access services, and security services interact with CPI-C to support programs and conversations. In addition, a resource recovery component provides its own programming interface and cooperates with CPI-C to coordinate changes to distributed resources, such as data and

Figure 2 Programs using CPI-C to converse through a network


PROGRAM A
PROGRAM B
CONVERSATION WITH PROGRAM C
CONVERSATION WITH PROGRAM D
CPI FOR COMMUNICATIONS
NETWORK
CRM X
LOGICAL CONNECTION
LOGICAL CONNECTION
LOGICAL CONNECTION
CRM Y
CPI FOR COMMUNICATIONS
CRM Z
PRODUCT-SPECIFIC INTERFACE
CONVERSATION WITH PROGRAM A
CONVERSATION WITH PROGRAM B
PROGRAM C
PROGRAM D
CRM = COMMUNICATION RESOURCE MANAGER
CPI = COMMON PROGRAMMING INTERFACE

files. These services generally provide usability features for the CPI-C programmer. They will be discussed further in the section on advanced concepts and services.

The underlying protocol support for CPI-C is provided by communication resource managers (CRMs). CRMs establish logical connections between themselves to support conversations between programs. LU 6.2 CRMs support CPI-C in an SNA network;[2] OSI TP CRMs support CPI-C in an OSI network.[6] Figure 2 shows a sample network with programs using CRMs to support conversations. Notice that Program B can communicate with Program D even though Program D is not written to the CPI-C interface.

Conversation characteristics, side information, and system services are mechanisms that CPI-C uses to hide complexity from programs. The basic conversation services are described next.

The starter set. CPI-C defines a starter set of calls that provide the basic functions needed to write distributed applications. Table 1 shows the starter set calls. Initialize_Conversation, Allocate, and Accept_Conversation are used to start the conversation; Send_Data and Receive are used for data

Figure 3 The Hello, World example

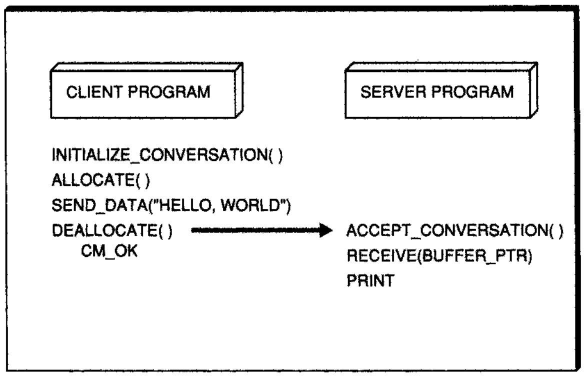


Table 1 The CPI-C starter set; these six calls provide enough function for many programs.

Call Pseudonym	Call Name
Initialize_Conversation	cminit
Allocate	cmallc
Accept_Conversation	cmaccp
Send_Data	cmsend
Receive	cmrcv
Deallocate	cmdeal

transfer; and Deallocate is used to end the conversation.

Each call has two names. One name is the actual name of the call, and the other is its pseudonym. For usability, CPI-C defines readable pseudonyms for the CPI-C calls, variables, characteristics, and values. Each CPI-C implementation provides pseudonym files that provide the standard set of pseudonyms. In C for example, the pseudonym file is named CPIC.H.

Establishing a conversation. To begin exchanging data, a pair of CPI-C programs must have an active conversation between them. The program initiating a conversation issues the Initialize_Conversation call specifying the symbolic destination name for the partner program. CPI-C allocates resources for the conversation, uses the symbolic destination name to retrieve destination information from side information, initializes the conversation characteristics, and returns a conversation identifier to the program. If the program wishes to change the destination information or the initial conversation characteristic values set by CPI-C, it may do so by issuing the appropriate Set calls before issuing the Allocate call. (We discuss use of a distributed directory for destination information in a later section.)

When the program issues the Allocate call, the local CRM sends a conversation startup request to the remote CRM carrying the name of the program partner, security tokens (if any), and the function level for the conversation. The remote system validates the security information, allocates resources for the conversation, and starts the partner program, if it is not already in execution.

The partner program accepts the conversation by issuing the Accept_Conversation call. CPI-C initializes the conversation characteristics for the end of the conversation for that partner program and returns a conversation identifier. The program may examine the conversation characteristics by issuing the appropriate Extract calls, and the programs can begin to exchange data.

Transferring data. CPI-C conversations can range from simple one-way messages to an extended exchange of multiple messages sent by both partners. The Send_Data and Receive calls are used to transfer data. By default, the local system accumulates data to be sent until it has a sufficient amount for transmission. The effect of this buffering technique is excellent network performance. The following example application demonstrates this feature.

A program issues the Send_Data call to send one data record to the remote program. On a half-duplex conversation, one program at a time has *send control*, which is the right to send data. Only that program may issue the Send_Data call. On a full-duplex conversation, both programs can send and receive data concurrently.

A program issues the Receive call to get information from its partner. The information received can be data, control information, conversation status, or other information. For example, a Receive call could return both data and send control to the program. The program can specify the amount of data to be received and the variable in which to receive it, and is notified of the actual length of data received. Further information about the conversation can be obtained by examining the values of the other parameters returned. See Table 2 for further details.

Table 2 Some important parameters on CPI-C calls.

conversation_ID	Complex programs can have many concurrent conversations. When a program successfully issues an Initialize_Conversation or Accept_Conversation call, CPI-C returns a unique identifier for the new conversation, called a conversation_ID. The program then supplies that conversation_ID as an input parameter on all subsequent calls related to that conversation.
control_information_received	The control_information_received parameter returns control information to the program. It can indicate that the remote program wants send control (on a half-duplex conversation), that expedited data are available to be received, or that the partner program has accepted or rejected a request for a conversation.
data_received and status_received	data_received and status_received are return parameters on the Receive call. The data_received parameter indicates whether data were actually received, and if so, whether a complete chunk of data (a record) was received. The status_received parameter helps guide the program regarding what to execute next. It may indicate that the program now has send control, or that the partner has issued a synchronization request, and whether the conversation is to be deallocated.
return_code	For every CPI-C call, CPI-C replies with a return code that indicates what happened. The return code denoting successful completion is CM_OK. Other return codes indicate specific errors.

Ending a conversation. Many housekeeping steps are similar among communicating CPI-C programs. One side issues Initialize_Conversation and Allocate calls; the other side issues Accept_Conversation. Similarly, some things are always done at the end of a CPI-C program.

To end a conversation, a program uses the Deallocate call. On a half-duplex conversation, the Deallocate call operates quickly. CPI-C returns to the issuing program, without waiting for the partner to acknowledge that it is ready to end the conversation. Only one side needs to issue a Deallocate call, which ends the conversation for both sides. We resume discussion of deallocating conversations later, including its use with full-duplex conversations and program synchronization.

Figure 3 illustrates the sequence of calls discussed previously. This example application has two programs: one to send the phrase "Hello, world," and one to receive the incoming phrase and display it. The originator program (the client) sends the string "Hello, world"; the target program (the server) receives and displays the string. The skeleton for these two programs is shown. C code for the client and server is shown in Appendix A. The arrow in Figure 3 represents that the message flows from the client program to the server program after the Deallocate. In this example the underlying flow of conversation data occurs after the last call on the client side, but before the first call on the server side. It is the Deallocate call by the client that causes all buffered information to flow. This illustrates how data transmission through a network is optimized. The server uses just two CPI-C calls: Accept_Conversation and Receive. When the server executes the Receive call, it gets the arriving data, as well as the notification that the conversation has already been deallocated. At that point, it simply prints what has arrived, and exits.

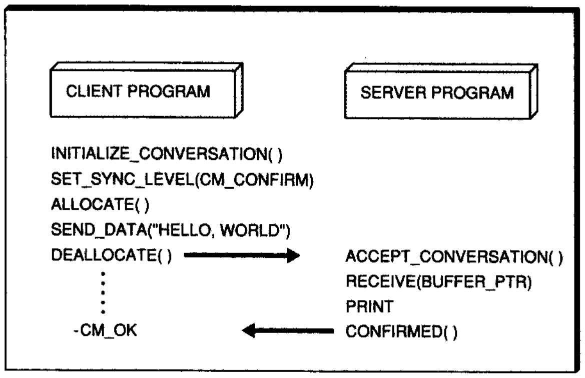
Advanced concepts and services

Beyond the starter set calls, CPI-C offers a diverse range of facilities to the programmer. These allow the programs to synchronize processing, notify the partners of errors, and use distributed services. Some of these features are outlined next.

Data transfer techniques. Four transfer techniques are important to mention next. With *immediate transfer*, the local system accumulates data to be sent to the remote system in its send buffer until it has a sufficient amount for transmission. A program can use the Flush call to empty the system send buffer and send the data to the partner immediately. This allows the partner to begin processing the data.

Programs using a *half-duplex* conversation must transfer send control back and forth for a two-way

Figure 4 Hello, World with confirmation example



data exchange. The receiving program can use the Request_To_Send call to request send control. The sending program issues the Prepare_To_Receive or Receive call to transfer send control.

A program wanting *full-duplex* data transfer sets the send_receive_mode characteristic to CM_FULL_DUPLEX prior to issuing the Allocate call. Programs that use a full-duplex conversation send and receive data concurrently.

Using *expedited data*, a program can send urgent data to its partner program using the Send_Expedited_Data and Receive_Expedited_Data calls. Expedited data may be delivered ahead of normal data sent earlier, and are guaranteed to be delivered ahead of any normal data sent after them. This function matches the "out-of-band" data function in TCP/IP.

Synchronization. The program that initializes a conversation can choose among four levels of synchronization by setting the sync_level conversation characteristic to CM_NONE, CM_CONFIRM, CM_SYNC_POINT, or CM_SYNC_POINT_NO_CONFIRM. Each level corresponds to a different degree of assurance about a partner's processing of a particular transaction.

The sync_level of CM_NONE allows the programs to communicate without any synchronization support from CPI_C. If the programs require any synchronization, they have to perform it using program logic.

Confirmation. The sync_level of CM_CONFIRM allows a program to request an acknowledgment from its partner. This exchange of a request and its acknowledgment is termed a *handshake*. Programs typically use the positive acknowledgment to indicate that all data have been successfully processed. This synchronization level is permitted only on half-duplex conversations. The program with send control issues the Confirm call to initiate the handshake. The partner is notified of the outstanding handshake by a value of CM_CONFIRM_RECEIVED on the status_received parameter of a Receive call. The partner then issues the Confirmed call for a positive acknowledgment, or a Send_Error or Deallocate call for a negative acknowledgment. The successful completion of the Confirm call, detected by a CM_OK return code, indicates that the partner responded with a positive acknowledgment. See the "Hello, World with confirmation example" in Figure 4 for the use of the confirmation level of synchronization.

In this example, confirmation logic is added to the prior Hello, World example, and this logic uses the sync_level of CM_CONFIRM. C code for the client and server programs is shown in Appendix A.

The client program issues the Set_Sync_Level call to choose this level of synchronization. Instead of issuing a separate Confirm, the program combines the function of the Confirm call in the Deallocate call. The Deallocate call uses the current value of sync_level, which is CM_CONFIRM, and does not complete until an acknowledgment is received. Note that the previous example used the default sync_level of CM_NONE.

Resource recovery. The sync_level of CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM allows programs to synchronize using *two-phase commit* protocols[14–17] accessible through a resource recovery interface. Currently, CPI-C supports the CPI-RR[18] and the X/Open TX[19] resource recovery interfaces. Figure 5 shows the interaction between various local components when this level of synchronization is used. A program issues the Commit call to the resource recovery interface to establish synchronization points in its processing. The processing and changes that occur to resources between two consecutive synchronization points are collectively referred to as a *transaction* (a logical unit of work in APPC terminology). If the underlying two-phase commit protocols, coordinated by the *transaction manager*, can make permanent all the changes to resources made by the program, then the Commit call is successful; otherwise, the transaction is rolled back to the latest successful synchronization point. If the program detects a failure, it issues a Backout call to explicitly roll back to the previous synchronization point.

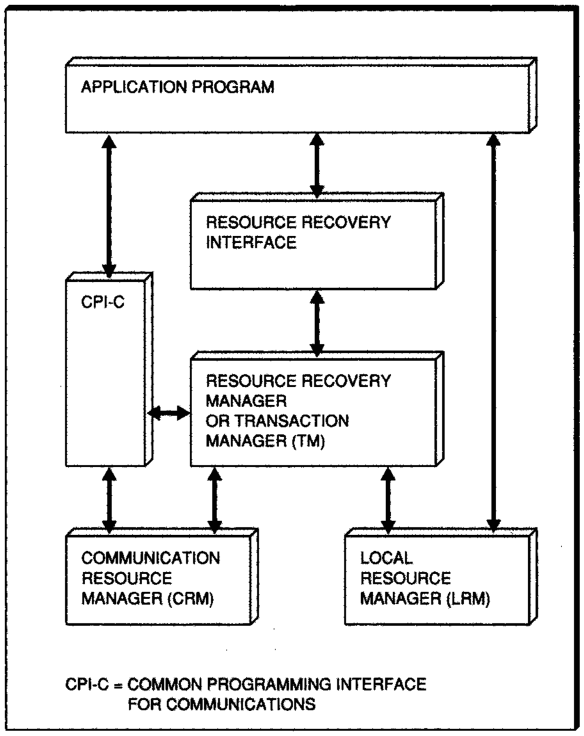
CPI-C cooperates with the transaction manager by passing synchronization information, which consists of take-commit and take-backout notifications, to the programs. When a program issues a Commit call, the partner program receives a take-commit notification via the status_received parameter of the Receive call. The partner program can now commit or backout. A take-backout notification is indicated by any of several return codes, such as CM_TAKE_BACKOUT or CM_DEALLOCATE_ABEND_BO, that indicate a Backout call issued by the partner or a system failure. After receiving such a notification, the program can issue a Backout call to roll back to the previous synchronization point, or deallocate all the conversations associated with the ongoing transaction abnormally.

Error notification. CPI-C provides an easy way for a program to notify its partner of an error. The error notification does not have to be sent as data using normal data transfer, as is the case for sockets programming. When a program issues the Send_Error call, the remote program receives a return code indicating that its partner has detected an error. If Send_Error is issued when data are being received, arriving data may be purged; the data sender is notified of the possible purging. When this call completes successfully on a half-duplex conversation, the local program has send control.

Concurrent operations and nonblocking support. Much effort has been devoted to allowing programs to continue productive work when a needed resource is temporarily not available. Operating systems have added support for multithreading and event management. A number of APIs provide specific mechanisms that allow the program to avoid being blocked waiting for a particular resource. For example, the *select* function in sockets programming allows a program to wait for activity across a number of file descriptors. Communication programs have the same requirements. A program serving multiple clients cannot afford to be blocked waiting for data from one client. CPI-C provides support for concurrent operations and nonblocking calls for such programs.

Concurrent operations. A program that executes on a system with multithreading support can use multiple threads for concurrent operations on one or more conversations. For this, CPI-C groups calls on a conversation into logical associations called *conversation queues*. Calls associated with different queues are processed independently, and concurrent calls may be in progress on multiple conversation queues. Only one call associated with a given conversation queue is allowed to be in progress at a time.

Figure 5 Components involved in distributed transaction processing


APPLICATION PROGRAM
RESOURCE RECOVERY INTERFACE
CPI-C
RESOURCE RECOVERY MANAGER OR TRANSACTION MANAGER (TM)
COMMUNICATION RESOURCE MANAGER (CRM)
LOCAL RESOURCE MANAGER (LRM)
CPI-C = COMMON PROGRAMMING INTERFACE FOR COMMUNICATIONS

What conversation queues are available on a conversation depends on whether the conversation is half-duplex or full-duplex. A half-duplex conversation has a send-receive queue for sending and receiving normal data. A full-duplex conversation has two queues for normal data: a send queue and a receive queue. Send and receive operations can be in progress simultaneously. Both types of conversation have an initialization queue (used only during conversation establishment) and two expedited data queues: the expedited-send and expedited-receive queues.

Nonblocking calls. A program can specify whether its calls should be processed in *blocking* or *non-*

blocking mode. When a call is processed in blocking mode, the program (or thread) is suspended until the call operation completes. For example, a Receive call processed in blocking mode does not return control until data or other information are available to be received. In nonblocking mode, if the call operation cannot complete immediately, the call gets the CM_OPERATION_INCOMPLETE return code. The call operation remains in progress as an *outstanding* operation. CPI-C provides two levels of nonblocking support: *conversation-level* and *queue-level* nonblocking.

When conversation-level nonblocking is used on a conversation, the program sets the processing mode for the conversation; the queues are ignored. The program can have at most one outstanding operation at a time on the conversation. The program issues the Wait_For_Conversation call to wait for completion of any outstanding operation across all its conversations using conversation-level nonblocking.

When queue-level nonblocking is used on a conversation, the program sets the processing mode on a queue basis. It can have multiple outstanding operations on the conversation, but only one per queue. The program can issue the Wait_For_Completion call to wait for completion of outstanding operations, specifying a list of outstanding operations. The call returns the list of those operations that have completed. The program may choose instead to specify a queue and a callback function to be invoked when an outstanding operation on that queue completes.

Support for servers and distributed services. Servers are increasingly important components within a network of distributed systems. CPI-C allows server programs to serve multiple clients efficiently. CPI-C also supports distributed directory and security services, which are becoming prevalent in networks.

Server support. A number of CPI-C functions are designed for server programs. A server can register multiple names with CPI-C to represent different services it provides to clients. A server can accept multiple incoming conversations and perform work for multiple clients simultaneously. The use of concurrent operations and nonblocking calls allows a server to support multiple clients efficiently. Within the operating environment, a *context manager* works with CPI-C to assist servers in managing the work done on behalf of multiple clients. Consider, for example, a server program with several clients. If the server initiates a conversation with another server on behalf of a particular client, the security tokens that accompany the conversation startup request should represent that client. The program, CPI-C, and the context manager work together to achieve this.

Each time a server accepts a new incoming conversation from a client, the context manager creates a new *context*, a collection of local attributes associated with the work done on behalf of that client. It contains attributes such as security information and an identifier for the transaction the client is processing. The context manager also maintains for the server a *current context*, the context within which the server is currently working. Attributes of the current context are used when the program takes certain context-sensitive actions, such as starting a new conversation for a client. A program may change the current context. In particular, the server in the example above should set its current context to that for the particular client prior to issuing the Allocate call.

Distributed directory. CPI-C initially provided two ways for a program to identify the partner program: side information and program-supplied information. Both methods have drawbacks. Side information supports only an eight-byte name space on the local system and must be administered on each system. Moving the partner program may result in updates to side information on multiple systems. Use of program-supplied information requires the program to use network-specific values, and the program may require recompilation if the address of the partner program changes. Use of a distributed directory addresses these problems and supports additional flexibility in identifying the partner program.

CPI-C programs can use information stored in an OSI X.500 directory,[20] an OSF DCE directory,[21] or any other directory supported by the CPI-C implementation. CPI-C defines a *program installation object*, a directory object that represents a single installation of a program. The object is identified in the directory by a distinguished name (DN). The object contains a program binding and optionally a program function identifier (PFID). A program binding contains the addressing and security information necessary for CPI-C to establish the conversation. A PFID, a unique identifier for the service available from the program, allows CPI-C to search for an appropriate partner based on the required service.

A program using destination information stored in a distributed directory can use the Set_Partner_ID call to supply a DN, a PFID, or a program binding to CPI-C. If the program supplies the DN, CPI-C uses it to retrieve the program installation object and extracts the program binding. If the program provides a PFID, CPI-C searches the directory to find a program installation object containing that PFID and extracts the program binding. (A system-specific default DN can limit the scope of the directory search.) If the program chooses to access the directory directly, it extracts the program binding from the directory object and passes it to CPI-C. CPI-C uses the program binding to establish the conversation.

Though use of the distributed directory eliminates the need for side information, administrative work is still needed. The network administrator must build a program installation object and add it to the directory when installing a program. The administrator must assign network addresses and security information, needed for the program binding, and a DN for the object according to network policies and naming conventions. The DN or PFID must be published or otherwise communicated to programs that want to initiate a conversation with the installed program. The advantage is that only the program binding must be updated if the program is moved and reinstalled.

Distributed security. Early versions of CPI-C supported a security system design that required a user to have a user identification (ID) and password on each system having resources to which the user wanted access. The user was required to manage multiple user IDs and passwords. Besides being administratively burdensome, the user ID and password flowed together and were subject to attack.

Now CPI-C includes support for a distributed security service that reduces the earlier deficiencies. In the new design, a user or system is defined once, with a principal name and password, to the security service. The user is authenticated once by a trusted authentication service, rather than by each system to which the user connects. The authentication service itself can be centralized or distributed. When the user's program initiates a conversation, the local CRM (communication resource manager) obtains encrypted authentication tokens from the authentication service and sends them with the conversation startup request. The remote CRM uses its local security service interface to validate the authentication tokens. Figure 6 shows a distributed security service interacting with CRMs to establish a conversation. The numbers 1–7 indicate the flow sequence.

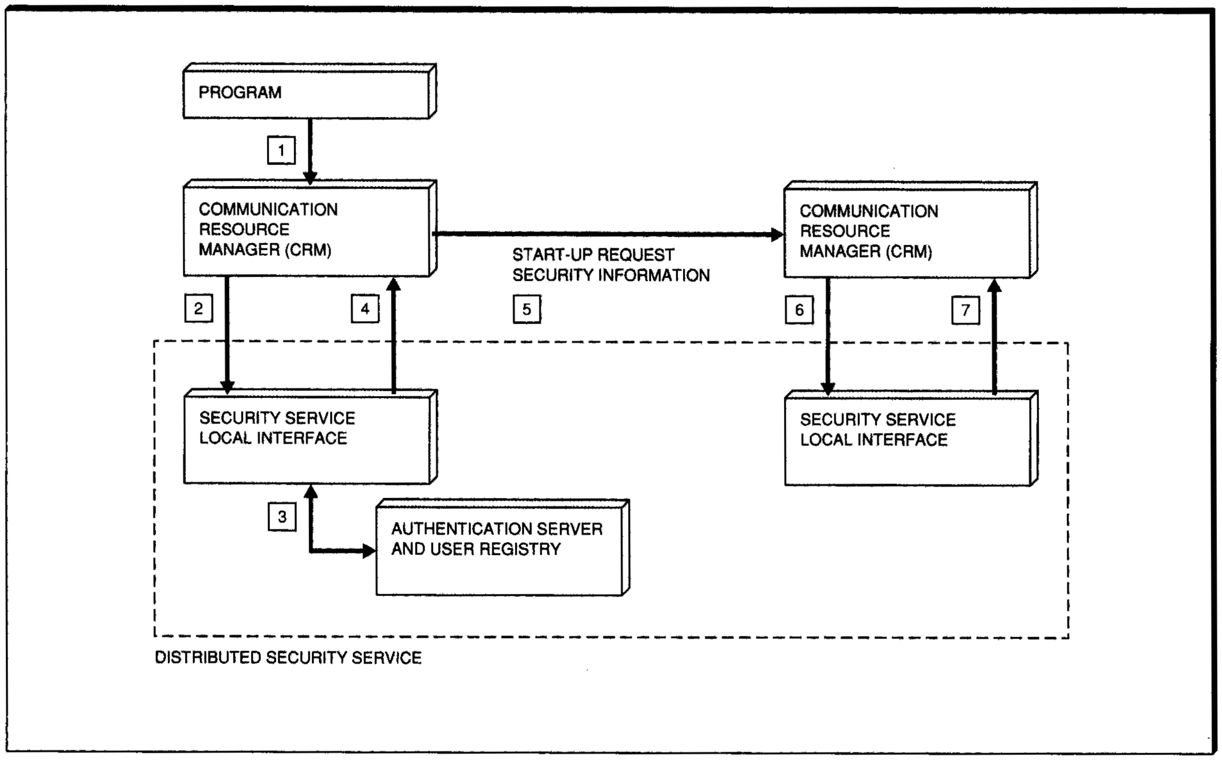
Two pieces of information required by the security server are obtained from the distributed directory: the principal name of the remote system and the required_user_name_type. The principal name is used by the security service to encrypt the security tokens that are transmitted through the network. The required_user_name is used by the local CRM to determine what type of user name should be sent with the conversation startup request.

Neither the user nor the user's program need be involved in security when a conversation is established if the initiating program uses the default value (CM_SECURITY_SAME) of the conversation_security_type characteristic. In this case, the system uses the security information from the program's current context to obtain the security tokens for the conversation startup request. A program with special requirements may specify that the remote system be authenticated before data are transmitted.

OSI TP support. CPI-C was initially defined to provide access to the services of APPC. When OSI TP became a standard in 1992,[6] with services closely patterned on those of APPC, CPI-C was mapped to the OSI TP services and extended to provide additional OSI TP support. CPI-C supports those features of OSI TP required to conform to all the OSI TP profiles defined by the standards body ISO/IEC (International Organization for Standardization/International Electrotechnical Commission). We describe next the features of CPI-C that are specific to supporting OSI TP.

When a program using an LU 6.2 CRM for a conversation within the scope of a transaction (and so using the resource recovery level of synchronization) commits that transaction, any further work done using that conversation is implicitly included as a part of the next transaction. This mode of running transactions is the *chained* mode. Sometimes, the user prefers the flexibility of using the same

Figure 6 CRM interaction with distributed security service


PROGRAM
1
COMMUNICATION RESOURCE MANAGER (CRM)
START-UP REQUEST SECURITY INFORMATION
5
COMMUNICATION RESOURCE MANAGER (CRM)
2
4
6
7
SECURITY SERVICE LOCAL INTERFACE
SECURITY SERVICE LOCAL INTERFACE
3
AUTHENTICATION SERVER AND USER REGISTRY
DISTRIBUTED SECURITY SERVICE

conversation for some unprotected work after the previous transaction commits but before a later transaction is initiated. This mode of running transactions is the *unchained* mode. Use of unchained transactions allows programs to exchange data and control information outside the scope of a transaction without having to use a different conversation. The OSI TP CRM supports both the chained and unchained modes, while the LU 6.2 CRM supports only the chained mode. (For further details of how a CRM reconciles these two modes, see Reference 22.) When a program is using unchained transactions, it specifies when the next transaction begins after the current transaction ends by issuing a tx_begin call to the X/Open TX interface.[19] Similarly, the program can use the Set_Transaction_Control call to specify whether a conversation is automatically included in the scope of the next transaction when the current transaction ends.

A program can send data, set by the Set_Initialization_Data call before allocating the conversation, on the conversation startup request. The program can also specify whether it wants an acknowledgment indicating whether its request for a conversation was accepted by the partner program.

Deallocating a conversation. A conversation is automatically deallocated when a program is notified of a system error or network failure on a return code; at other times, either partner program can choose to deallocate the conversation. In the latter case, the program can set the deallocate_type characteristic to indicate how the conversation is to be deallocated, before issuing the Deallocate call.

A program deallocates the conversation by issuing the Deallocate call, and the partner is subsequently notified. Depending on the current values of the deallocate_type and sync_level characteristics, the conversation may be deallocated after synchronization with the partner has been performed.

If no synchronization with the partner is to be performed, a half-duplex conversation is deallocated when the Deallocate call completes. The Hello, World example shows deallocation with no synchronization. For a full-duplex conversation, the

Deallocate call applies to the outgoing direction of data transfer only. When a program issues Deallocate, it can no longer send data, and the partner program can no longer receive data. When both programs have issued Deallocate, the conversation is deallocated.

Data mapping, improved server support, and object-oriented programming are likely extensions to CPI-C.

If either the confirmation or the resource recovery level of synchronization with the partner is to be performed, the conversation is deallocated only after the appropriate synchronization call is issued and completes successfully. If the synchronization call does not complete successfully, the conversation is not deallocated; it remains active for further processing. The Hello, World with confirmation example shows deallocation with the confirmation level of synchronization.

The program may choose to deallocate the conversation abnormally when it detects an error condition that prevents it from continuing normal processing. This type of deallocation can be performed at any time. Any data in transit to the program that issues the call are purged. The partner program is notified of the abnormal deallocation by a CPI-C call return code such as CM_DEALLOCATE_ABEND.

Problem determination. Many CPI-C implementations provide extensive trace facilities for problem determination during development and execution of CPI-C programs, and CPI-C can supply secondary information about the condition that caused a failure. Return codes indicate the outcome of a CPI-C call and allow the program to determine what action to take. They are not sufficient for problem determination; in some cases, a number of error conditions can yield the same return code. When a call fails, the program may obtain secondary information for use in problem determination by issuing the Extract_Secondary_Information call. Secondary information contains a condition code, a description of the condition, the cause of the condition, suggested actions, and any additional information supplied by the implementation.

Conformance classes

An important aspect of a standard is ensuring consistency in implementation and providing a means of describing what has been implemented. The ISO/IEC standards body addresses this issue by defining *profiles*, subset descriptions for consistent implementation across systems, for various international standards, and the X/Open consortium defines *conformance statement questionnaires* to be provided by implementers of its APIs. Similarly, the CIW (CPI-C Implementers' Workshop) has defined a set of *conformance classes* to foster an orderly marketplace for implementation, product selection, and use of CPI-C. The conformance classes are used in product announcements, requirement specifications, and procurement specifications.

The definition specifies one mandatory conformance class (conversations) and a set of optional conformance classes. The mandatory class contains function that allows a program to start and end half-duplex conversations, exchange data, use confirmation and error notification, and set and modify conversation characteristics. Optional conformance classes include recoverable transactions, full-duplex, queue-level nonblocking, and directory. As an example, the IBM Operating System/400* (OS/400*) product implements the conversations, LU 6.2, recoverable transactions, security, and data conversion routines conformance classes. For more information on IBM product support of conformance classes, see Reference 23.

Future extensions

Extensions to CPI-C are being defined in the CIW. Data mapping, improved server support, and object-oriented programming are three likely extensions.

The distributed processing environment encompasses heterogeneous systems with different data representations, such as EBCDIC and ASCII for character data. As a result, data flowing between systems may require conversion. Data mapping support will allow programs to instruct CPI-C to use conversion routines to convert user data.

The Accept_Conversation call currently accepts an incoming conversation for any name registered by the program. Improved server support will enable a program to create separate threads, each of which can accept incoming conversations for a specific name associated with the particular service offered by that thread.

CPI-C is currently a procedural API: programs issue subroutine calls to CPI-C library routines. The definition of a CPI-C object class library will allow programs to use object-oriented programming techniques with CPI-C.

Conclusion

CPI-C is an evolving programming interface for program-to-program communication. From its definition and initial development by IBM in 1988, it has been extended to meet new requirements and to respond to industry trends and advances in technology. The latest version, CPI-C 2.0, was published in June 1994, by the CPI-C Implementers' Workshop.

CPI-C has become a standard interface for conversations—standard across languages, protocols, and systems. Wide implementation facilitates program portability, and ease-of-use features enable high programmer productivity. With a rich set of conversation services and extensions to meet new requirements under development, CPI-C is poised to serve distributed applications well into the future.

Acknowledgment

The authors gratefully acknowledge the assiduous support of Gary Schultz during the preparation of this paper. His editorial suggestions and those of the anonymous reviewers contributed greatly to the improvement of its structure and presentation.

Appendix A. C source code examples

The following examples and discussion are derived from Reference 24, which contains many other CPI-C programming examples as well. Portions of this appendix are reproduced with permission from McGraw-Hill, New York.

The source code files for our two programs are named HELLO.C and HELLOD.C. We are using a naming convention that has been adopted by UNIX** programmers. The server side (the side sitting and waiting for something to do) is referred to as the "daemon." Its name is constructed by adding a D to the name of its partner, the client program. Thus, the client's name is HELLO and server's name is HELLOD. The two programs each comprise basic logic shown in **boldface**, and confirmation logic shown in plain font. Two pairs of partner programs are formed by either omitting or using the confirmation logic.

You will see that every parameter on every call is a pointer. CPI-C calls only by reference, which lets it work the same across all programming languages.

The HELLO source code with the confirmation logic is a good starting place for someone who wants to start writing a CPI-C application. With confirmation processing, a programmer can ensure that the server program actually starts and runs properly. If this runs, then everything has been set up properly between the two programs.

The Hello, World client. We will show the source code for the Hello, World client first. The reader will see that without confirmation, it consists of four boldface CPI-C calls and a return statement. The four CPI-C calls are the same four shown in the Hello, World examples earlier.

CPI-C call parameters either supply input to CPI-C as part of the call, or get output information back from CPI-C. For example, the first call in HELLO is Initialize_Conversation(). It has three parameters: conversation_ID, symbolic_destination_name, and epic_return_code, and looks like this:

```
cminit(
     /* Initialize_Conversation          */
   conversation_ID,
     /* O: returned conversation ID   */
   SYM_DEST_NAME,
     /* I: symbolic destination name   */
   &cpic_return_code);
     /* O: return code from this call   */
```

The conversation_ID parameter is an output parameter; it points to a field into which CPI-C will return information.

The symbolic_destination_name is an input parameter; it points to the name that CPI-C uses to decide who and where the partner program is. The symbolic destination name is the CPI-C method of letting one say to whom one wants to talk. The field always is eight characters long, so if the name is less than eight characters long, it needs to be padded on the right with blanks. The SYM_DEST_NAME information is located in the CPI-C side information file discussed previously.

The cpic_return_code parameter is an output parameter; it points to a field where CPI-C will write an integer that represents the return code from the call.

The following example client program is named HELLO.C. The basic program (in **boldface**) contains two CPI-C calls to set up the conversation: one call to send the data, and one call to take down the conversation. The plain font portions add confirmation logic.

```
/*-------------------------------------------------------------------
  * CPI-C "Hello, world" program.
  * Code sample (Client side (file HELLO.C))
  * Example modified to fit page
* ------------------------------------------------------------------*/
#include <cpic.h>
        /* conversation API library              */
#include <string.h>
        /* strings and memory                    */
#include <stdlib.h>
        /* standard library                      */

/* this hardcoded sym_dest_name is 8 chars
   long & blank padded                           */
#define SYM_DEST_NAME
  (unsigned char*)"HELLO2S"

/* this is the string we're sending to the partner */
#define SEND_THIS
  (unsigned char*)"Hello, world"

int main(void)
{
  unsigned char
    conversation_ID[CM_CID_SIZE];
  unsigned char
    *    data_buffer = SEND_THIS;
  CM_INT32
    send_length =
    (CM_INT32)strlen(SEND_THIS);
  CM_RETURN_CODE   cpic_return_code;
  CM_SYNC_LEVEL
    sync_level = CM_CONFIRM;

  CM_REQUEST_TO_SEND_RECEIVED
    rts_received;

  cminit(
      /* Initialize_Conversation                 */
    conversation_ID,
      /*   O: returned conversation ID           */
    SYM_DEST_NAME,
      /*   I: symbolic destination name          */
    &cpic_return_code);
      /*   O: return code from this call         */

  cmssl(
      /* Set_Sync_Level                          */
    conversation_ID,
      /*   I: conversation ID                    */
    &sync_level,
      /*   I: set sync_level to CONFIRM          */
    &cpic_return_code);
      /*   O: return code from this call         */

  cmallc(
      /* Allocate                                */
    conversation_ID,
      /*   I: conversation ID                    */
    &cpic_return_code);
      /*   O: return code from this call         */

  cmsend(
      /* Send_Data                               */
    conversation_ID,
      /*   I: conversation ID                    */
    data_buffer,
      /*   I: send this buffer                   */
    &send_length,
      /*   I: length to send                     */
    &rts_received,
      /*   O: was RTS received?                  */
    &cpic_return_code);
      /*   O: return code from this call         */

  cmdeal(
      /* Deallocate                              */
    conversation_ID,
      /*   I: conversation ID                    */
    &cpic_return_code);
      /*   O: return code from this call         */

  return(EXIT_SUCCESS);
}
```

The Hello, World server. The code in the basic server, HELLOD, contains just two starter set CPI-C calls. Accept_Conversation() gets a conversation ID for the server side. The Receive() call gets the arriving data, as well as the notification that the conversation has already been deallocated. (Again, the plain font adds the confirmation logic.)

We set aside a data_buffer to receive into that we have arbitrarily made 101 bytes long. On the Receive(), we set the requested length to only 100 bytes. In case we receive exactly 100 bytes, we want to have room to append a "\0" on the end, so we can use printf() to display it.

Also, notice that we have added a call to the C getchar() routine on the server side. On most computers, the server program can be automatically started, pop up in a window, receive the string from the client and call printf(), then quickly close the window. By calling getchar(), the server program will at least wait for a user to press a key before it vanishes.

The following example server program is named HELLOD.C. The basic program (in **boldface**) includes just two CPI-C calls and a call to printf and getchar. The plain font portions are for the added confirmation logic. The Confirmed() call is issued, assuming the client issued a corresponding Confirm().

```
/*-------------------------------------------------------------------
  * CPI-C "Hello, world" program.
  * Code Sample (Server side (file HELLOD.C))
  * Example modified to fit page
* ------------------------------------------------------------------*/
#include <cpic.h>
      /* conversation API library                             */
#include <stdio.h>
      /* file I/O                                             */
#include <string.h>
      /* strings and memory                                   */
#include <stdlib.h>
      /* standard library                                     */

int main(void)
{
  unsigned char
    conversation_ID[CM_CID_SIZE];
  unsigned char   data_buffer[100+1];
  CM_INT32        requested_length =
    (CM_INT32)sizeof(data_buffer)-1;
  CM_INT32        received_length = 0;
  CM_RETURN_CODE  cpic_return_code;

  CM_DATA_RECEIVED_TYPE data_received;
  CM_STATUS_RECEIVED     status_received;
  CM_REQUEST_TO_SEND_RECEIVED
    rts_received;

  cmaccp(
      /* Accept_Conversation                                  */
    conversation_ID,
      /*   O: returned conversation ID                        */
    &cpic_return_code);
      /*   O: return code from this call                      */

  cmrcv(
      /* Receive                                              */
    conversation_ID,
      /*   I: conversation ID                                 */
    data_buffer,
      /*   I: where to put received data                      */
    &requested_length,
      /*   I: maximum length to receive                       */
    &data_received,
      /*   O: data complete or not?                           */
    &received_length,
      /*   O: length of received data                         */
    &status_received,
      /*   O: has status changed?                             */
    &rts_received,
      /*   O: was RTS received?                               */
    &cpic_return_code);
      /*   O: return code from this call                      */

  data_buffer[received_length] = '\0';
      /* insert a null                                        */
  (void)printf("%s\nPress a key to end the
    program . . . \n", data_buffer);

  cmcfmd(
      /* Confirmed                                            */
    conversation_ID,
      /*   I: conversation ID                                 */
    &cpic_return_code);
      /*   O: return code from this call                      */

  (void)getchar();
      /* pause for any keystroke                              */

  return(EXIT_SUCCESS);
}
```

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of X/Open Co., Ltd.

Cited references

1. J. P. Gray, P. J. Hansen, P. Homan, M. A. Lerner, and M. Pozefsky, "Advanced Program-to-Program Communication in SNA," *IBM Systems Journal* **22**, No. 4, 298–318 (1983).
2. *SNA LU 6.2 Peer Protocols Reference,* SC31-6808, IBM Corporation (1988); available through IBM branch offices.
3. *Systems Network Architecture Transaction Programmer's Reference Manual for LU Type 6.2,* GC30-3084, IBM Corporation; available through IBM branch offices.
4. J. Sanders, M. Jones, J. Fetvedt, and M. Ferree, "A Communications Interface for Systems Application Architecture," *Selected Areas in Communications, IEEE Journal* **7**, No. 7, 1073–1081 (Sept 1989).
5. *X/Open Developers' Specification: CPI-C,* X/Open Company Limited, Apex Plaza, Forbury Road, Reading, Berksire, RG1, 1AX UK (1990).

6. *Information Technology—Open Systems Interconnection-Distributed Transaction Processing—Part 1: OSI TP Model; Part 2: OSI TP Service,* ISO/IEC JTC 1/SC 21N, ISO Central Secretariat, 1 rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland (April 1992).
7. D. Pozefsky, R. Turner, A. K. Edwards, S. Sarkar, J. Mathew, G. Bollella, K. Tracey, D. Poirier, J. Fetvedt, W. S. Hobgood, W. A. Doeringer, and D. Dykeman, "Multiprotocol Transport Networking: Eliminating Application Dependencies on Communications Protocols," *IBM Systems Journal* **34**, No. 3, 472–500 (1995, this issue).
8. *X/Open CAE Specification: CPI-C,* X/Open Company Limited, Apex Plaza, Forbury Road, Reading, Berksire, RG1, 1AX UK (1992).
9. *Common Programming Interface Communications CPI-C 2.0 Specification,* SC31-6180, IBM Corporation (1994); available through IBM branch offices.
10. *Distributed Transaction Processing: The CPI-C Specification, Version 2, X/Open Preliminary Specification,* X/Open Company Limited, Apex Plaza, Forbury Road, Reading, Berksire, RG1, 1AX UK (1994).
11. M. L. Hess, J. A. Lorrain, and G. R. McGee, "Multiprotocol Networking—a Blueprint," *IBM Systems Journal* **34**, No. 3, 330–346 (1995, this issue).
12. A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* **2**, No. 1, 39–59 (Feb. 1984).
13. B. Blakeley, H. Harris, and R. Lewis, *Messaging and Queuing Using the MQI: Concepts and Analysis, Design and Development,* McGraw-Hill, Inc., New York (May 1995).
14. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley Publishing Co., Reading, MA (1987).
15. J. N. Gray, "Notes on Database Operating Systems," *Operating Systems—An Advanced Course,* R. Bayer, R. Graham, and G. Seegmuller, Editors, Lecture Notes on Computer Science, Vol. 60, Springer-Verlag, Inc., New York (1978).
16. B. W. Lampson, "Atomic Transactions," *Distributed Systems: Architecture and Implementation—An Advanced Course,* B. W. Lampson, Editor, Lecture Notes on Computer Science, Vol. 105, 246–265, Springer-Verlag, Inc., New York (1981).
17. *Systems Network Architecture Sync Point Services Architecture Reference,* SC31-8134, IBM Corporation (1994); available through IBM branch offices.
18. *Systems Application Architecture Common Programming Interface—Resource Recovery,* SC31-6821, IBM Corporation (1991); available through IBM branch offices.
19. *X/Open Distributed TP: a) The TX Specification, b) The XA Specification, c) The XA+ Specification,* X/Open Consortium, Apex Plaza, Forbury Road, Reading, Berksire, RG1, 1AX UK (November 1992, February 1992, April 1993).
20. *Information Technology—Open Systems Interconnect—The Directory: Overview of Concepts, Models, and Services,* ISO 9594-1, CCITT X.500, ISO Central Secretariat, 1 rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland.
21. *Open Software Foundation (OSF) Distributed Computing Environment (DCE) 1.0: Introduction to DCE,* OSF, ISO Central Secretariat, 1 rue de Varembe, Case postale 56, CH-1211 Geneva 20, Switzerland (March 1992).
22. G. Samaras, A. Kshemkalyani, A. Citron, "Reconciling Communication Protocol Support for Chained and Unchained Transactions," *Proceedings 2nd International Conference on Computer Applications to Engineering Systems, Cyprus* (July 1993), pp. 238–244.
23. *Common Programming Interface Communications CPI-C Reference Version 2.0,* SC26-4399, IBM Corporation; available through IBM branch offices.
24. J. Q. Walker II and P. J. Schwaller, *CPI-C Programming in C: An Application Developer's Guide to APPC,* McGraw-Hill, New York (1994).

Accepted for publication March 13, 1995.

Wendy S. Arnette *IBM Networking Software Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: wsa@vnet.ibm.com).* Ms. Arnette is an SNA networking advisor in the APPC Market Enablement group at IBM. Currently she provides technical and marketing support and education for both customers and vendors on APPC, APPN, and CPI-C. She also provides press and consultant relations for SNA and APPN on IBM's networking software products. Previously, Ms. Arnette worked on converting LU2 applications to APPC in IBM Systems Network Architecture. She has also published several APPC and APPN articles and papers. Ms. Arnette received a B.S. degree in computer science from North Carolina State University.

Ajay D. Kshemkalyani *IBM Networking Software Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: ajayk@vnet.ibm.com).* Dr. Kshemkalyani received the B. Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, India, in 1987, and the M.S. and Ph.D. degrees in computer and information science from the Ohio State University, Columbus, in 1988 and 1991, respectively. He is currently an advisory programmer in the Networking Systems Architecture area in IBM at Research Triangle Park and an adjunct assistant professor at North Carolina State University. He is presently involved with the networking broadband systems architecture at IBM. His current research interests include distributed systems, operating systems, computer architecture, and databases. He is a member of the ACM and the IEEE Computer Society.

Wayne B. Riley *IBM Networking Software Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: wriley@vnet.ibm.com).* Mr. Riley has been involved in the distributed applications arena since 1986. He has led APPC client/server application development projects for Nabisco Brands, Dun & Bradstreet, and University of North Carolina Hospitals. He joined IBM in August 1992 and is currently designing object-oriented distributed agents utilizing APPC. Mr. Riley received a B.S. degree from Fairleigh Dickinson University in Madison, New Jersey.

Jack P. Sanders *IBM Networking Software Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: jsanders@vnet.ibm.com).* Dr. Sanders is a senior engineer at IBM's Research Triangle Park facility. Prior to his current assignment in product development, he served as chair

of the CPI-C Implementers' Workshop and led the development of CPI-C 2.0 to a successful conclusion. His earlier work included managing the LU 6.2 Architecture department and contributing to the development of Systems Network Architecture. He earned a Ph.D. in mathematics from the University of Virginia and enjoyed an academic career before joining IBM in 1981.

Peter J. Schwaller *Ganymede Software Inc., 2 Davis Drive, Suite 124, Research Triangle Park, North Carolina 27709 (electronic mail: pjs@ffmail.nctda.org).* Mr. Schwaller joined IBM in 1989 to work on APPN architecture and later was a key member of the development team for IBM Networking Services/2. Most recently, he developed the CPI-C Sample Programs Toolkit available on CompuServe. Mr. Schwaller received a B.S.E. degree in electrical engineering and computer science from Duke University. He has recently completed a new book with John Q. Walker called *CPI-C Programming in C,* available from McGraw-Hill, Inc.

J. Chuck Terrien *IBM Networking Software Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: chuck@vnet.ibm.com).* Mr. Terrien joined IBM in 1984 as a member of the Office Systems Product Assurance area. From there he moved to the Network Management Architecture area, where he was instrumental in developing an architectural strategy to address integrated SNA and OSI network management. In 1989 he became a manager in the NetView Systems Test area, which provided testing for products such as NetView, NetView/PC, and DCAF. Mr. Terrien is currently in APPC Market Enablement, a technical marketing group that supports IBM customers, IBM products, and other APPC/APPN vendors. One of his responsibilities in APPC Market Enablement was managing the CPI-C architecture team. Mr. Terrien received a B.A. degree in business administration from Michigan State University and an M.S. degree in systems science from the State University of New York at Binghamton.

John Q. Walker *Ganymede Software Inc., 2 Davis Drive, Suite 124, Research Triangle Park, North Carolina 27709 (electronic mail: johnq@ganymede.nctda.org).* Dr. Walker is the Vice President for Development at Ganymede Software Inc., a company specializing in network software tools. In recent years, he managed departments in IBM networking and was responsible for network management, architecture, and software development. He helped found IBM's APPC market enablement team. In the early 1980s, he was a designer at IBM for the token-ring local area network, serving as editor for the IEEE 802.5 standard. Dr. Walker is the author of numerous articles and is a widely-traveled speaker on programming and networking technology. He recently completed a book with Peter Schwaller for McGraw-Hill, Inc., entitled *CPI-C Programming in C.* Dr. Walker received a B.S. in mathematics and a B.A. in music from Southern Illinois University. He also received an M.S. in computer science from Southern Illinios University, and a Ph.D. in computer science from the University of North Carolina.

Reprint Order No. G321-5579.