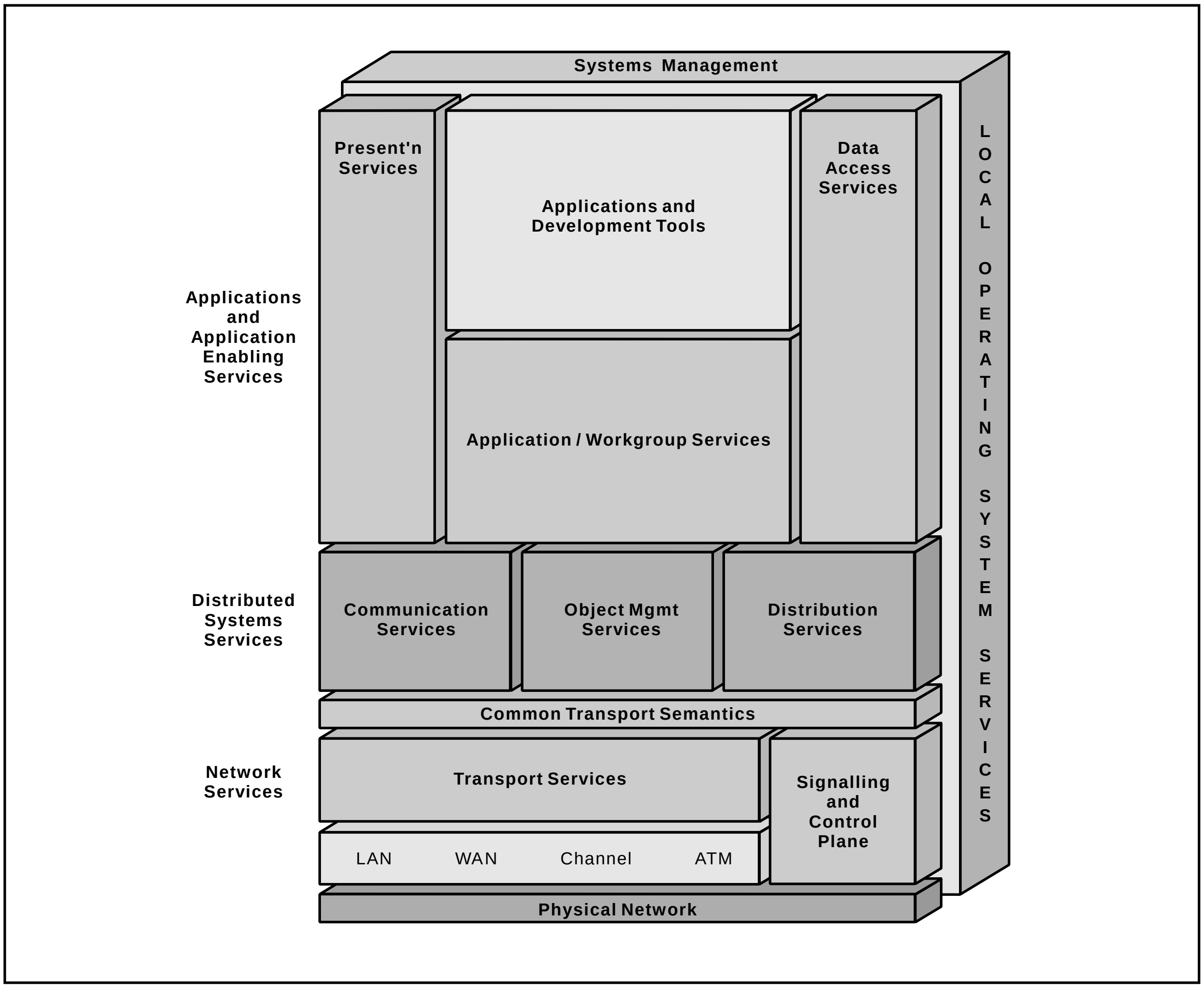
Open Blueprint

Conversational Resource Manager



Systems Management
Present'n Services
Applications and Development Tools
Data Access Services
Applications and Application Enabling Services
Application / Workgroup Services
LOCAL OPERATING SYSTEM SERVICES
Distributed Systems Services
Communication Services
Object Mgmt Services
Distribution Services
Common Transport Semantics
Network Services
Transport Services
Signalling and Control Plane
LAN
WAN
Channel
ATM
Physical Network

GC23-3914-01

Open Blueprint

Conversational Resource Manager

GC23-3914-01

About This Paper

Open, distributed computing of all forms, including client/server and network computing, is the model that is driving the rapid evolution of information technology today. The Open Blueprint structure is IBM's industry-leading architectural framework for distributed computing in a multivendor, heterogeneous environment. This paper describes the Conversational resource manager component of the Open Blueprint and its relationships with other Open Blueprint components.

The Open Blueprint structure continues to accommodate advances in technology and incorporate emerging standards and protocols as information technology needs and capabilities evolve. For example, the structure now incorporates digital library, object-oriented and mobile technologies, and support for internet-enabled applications. Thus, this document is a snapshot at a particular point in time. The Open Blueprint structure will continue to evolve as new technologies emerge.

This paper is one in a series of papers available in the *Open Blueprint Technical Reference Library* collection, SBOF-8702 (hardcopy) or SK2T-2478 (CD-ROM). The intent of this technical library is to provide detailed information about each Open Blueprint component. The authors of these papers are the developers and designers directly responsible for the components, so you might observe differences in style, scope, and format between this paper and others.

Readers who are less familiar with a particular component can refer to the referenced materials to gain basic background knowledge not included in the papers. For a general technical overview of the Open Blueprint, see the *Open Blueprint Technical Overview*, GC23-3808.

Who Should Read This Paper

This paper is intended for audiences requiring technical detail about the Conversational Resource Manager in the Open Blueprint. These include:

- Customers who are planning technology or architecture investments
- Software vendors who are developing products to interoperate with other products that support the Open Blueprint
- Consultants and service providers who offer integration services to customers

Contents

Figures

Summary of Changes

This revision includes:

- A description of CPI-C 2.1 functions
- A description of data conversion support
- A reference to alternative conversational semantics

Conversational Resource Manager

This paper describes the Open Blueprint resource manager for conversational communications (CCRM) and the Common Programming Interface for Communications (CPI-C), which is a standard application programming interface (API) for the CCRM functions.

The Conversational Model and CPI-C

The conversational model of program-to-program communication is commonly used in the industry today, and a wide variety of applications are based on this model. The model is described in terms of two applications that are speaking and listening—hence, the term *conversation*. A conversation is a logical connection between two programs that allows the programs to communicate with each other. The conversational model is implemented by the upper layer communications protocol, advanced program-to-program communication (APPC). APPC is also referred to as logical unit type 6.2 (LU 6.2).

Common Programming Interface-Communications (CPI-C) is a cross-system-consistent and easy-to-use programming interface for applications that require program-to-program communication. From an application's perspective, CPI-C provides the function necessary to enable this communication.

CPI Communications (CPI-C) defines a single programming interface to the underlying network protocols across many different programming languages and environments. The interface's rich set of programming services shields programs from the details of system connectivity and eases the integration and porting of application programs across supported environments.

Levels of CPI-C

CPI-C is an evolving interface, embracing functions to meet the growing demands from different application environments and achieve openness as an industry standard for communications programming. CPI-C was first introduced in 1987. Since then it has had five major function additions. The most recent levels are CPI-C 2.0, X/Open CPI-C 2.0, and CPI-C 2.1.

CPI-C is an API for conversational communications semantics and includes many functions to make programming distributed applications easier. The single mandatory function is support of conversations. All other functions are optional. It is not expected that an implementation of CPI-C will support all of the optional functions - only those that are appropriate for the target platform and environment. CPI-C has been standardized by the CPI-C Implementation Workshop (CIW)[1] and by X/Open.

CPI-C Functions Before 2.0

The following describes the functions provided by CPI-C in versions prior to 2.1.

- A common API for the following programming languages: Application Generator (CSP), C, COBOL, FORTRAN, PL/I, REXX and RPG
- The ability to start and end conversations
- Support for program synchronization through confirmation flows
- Error processing
- The ability to optimize conversation flow (using the CPI-C Flush and Prepare_To_Receive calls)
- Support for transaction managers

- Automatic parameter conversion
- Interoperability with programs using product-specific protocol-boundary interfaces.
- Local/remote transparency
- Support for non-blocking
- Support for data conversion
- Support for security parameters
- The ability to accept multiple conversations

CPI-C 2.0 Functions

CPI-C 2.0, which was completed by the CIW in 1994, provides enhancements to prior functions and offers new functions:

- Support for full-duplex conversations and expedited data
- Enhanced support for non-blocking processing with the addition of queue-level processing and a callback function
- Support for OSI-TP applications
- Support for use of a distributed directory
- Support for use of a distributed security service
- Support for secondary information to determine the cause of a return code
- Definition of conformance classes

X/Open CPI-C 2.0 Functions

X/Open CPI-C 2.0 enhances and updates X/Open CPI-C to be equivalent to CPI-C 2.0 with the following exceptions:

- Supports only C and COBOL programming languages.
- Does not include distributed directory support. Specifically, the Set_Partner_ID and Extract_Partner_ID calls are not included.
- For COBOL, X/Open uses COMP-5 for integers; CIW CPI-C 2.0 uses COMP-4 for integers.

CPI-C 2.1 Functions

CPI-C 2.1, which was completed by the (CIW) in 1995, provides the following functions:

- Support for automatic data conversion
- Support for the ability to accept conversations with a specific TP name
- Support for the specification of service TP names
- Support for additional return control options
- Enhancements to the Cancel_Conversations call to allow determination of cancelled operations

Communication across a Network

Figure 1 below illustrates the logical view of a sample network. It consists of three conversational communication resource manager[2] (CCRM) instances: CCRM X, CCRM Y, and CCRM Z. Each CCRM has two logical connections with two other CCRMs; the logical connections are shown as the gray portions of Figure 1 and enable communication among the CCRMs. The network shown in Figure 1 is a simple one. In a real network, the number of CCRMs and logical connections between the CCRMs can be in the tens of thousands or higher.

The CCRMs and the logical connections shown in Figure 1 are generic representations of real networks. In an APPC network, the CCRMs are referred to as *logical units* of type 6.2 and the logical connections as *sessions*. The physical network, which consists of nodes (processors) and data links between nodes, is not shown in Figure 1 because a program using CPI-C is not aware of these resources. A program uses the logical network of CCRMs, which in turn communicates with and uses the physical network.

The Open Blueprint structure supports the ability of CCRMs to communicate with non-open implementations of APPC. Figure 1 illustrates this by showing a conversation (between Programs B and D) where one program is using an implementation-specific protocol boundary interface. The fact that Program D is not using CPI-C is transparent to Program B.

Using CPI-C does not constrain the underlying transport network. Using Open Blueprint Common Transport Semantics, the CCRM's conversational protocols can be transmitted using any transport network, for example, Advanced Peer-To-Peer Networking (APPN), TCP/IP or NetBIOS. (For additional information about Common Transport Semantics, see the *Network Services* component description paper.)

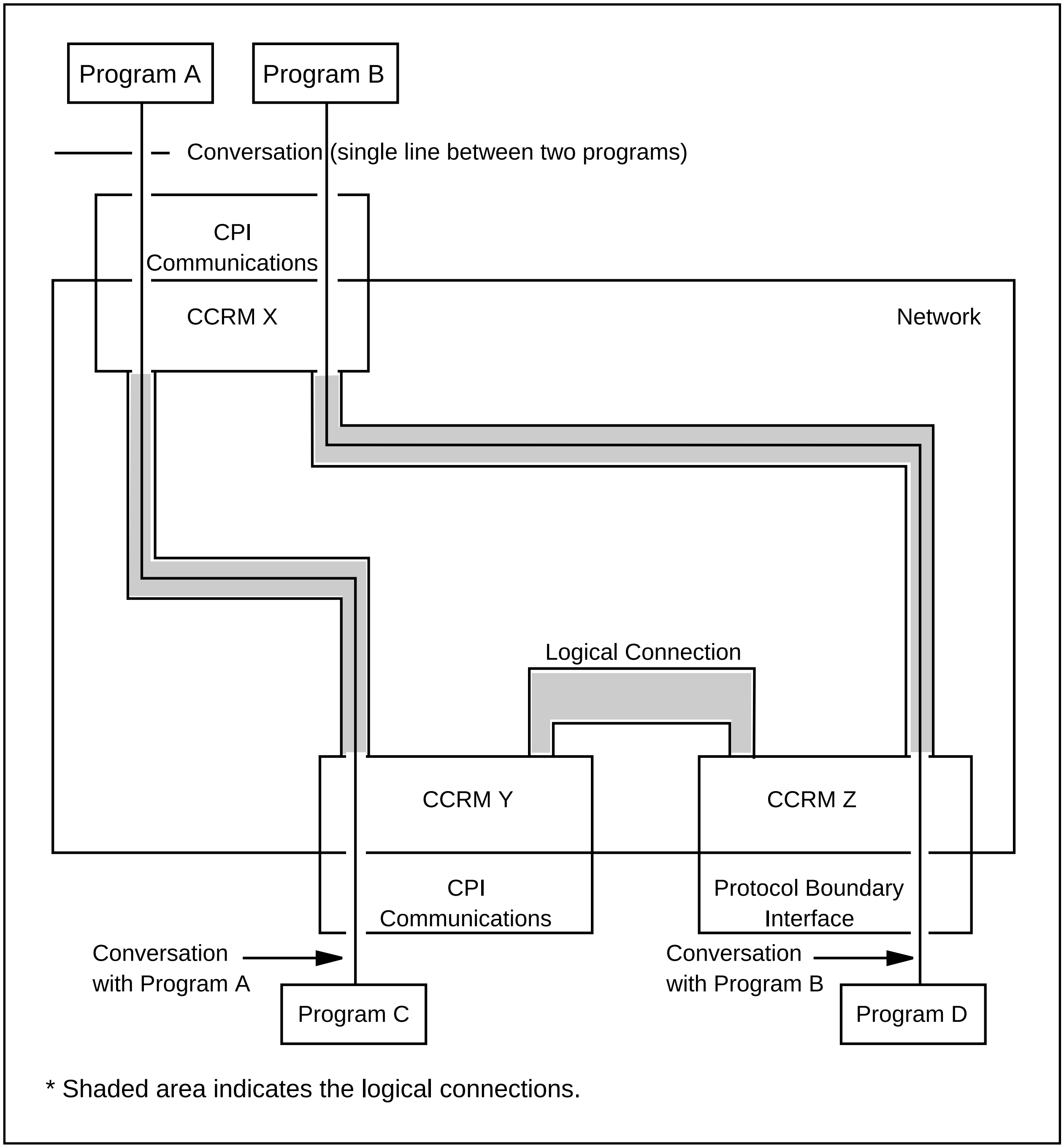


Figure 1. Programs Using CPI-C to Converse through a Network

Operating Environment

Figure 2 below gives a more detailed view of Program A's operating environment. As in Figure 1, Program A has established a conversation with its partner program (Program C). The line between the program and the CCRM represents Program A's use of CPI-C and CCRM specific calls to communicate with the CCRM.

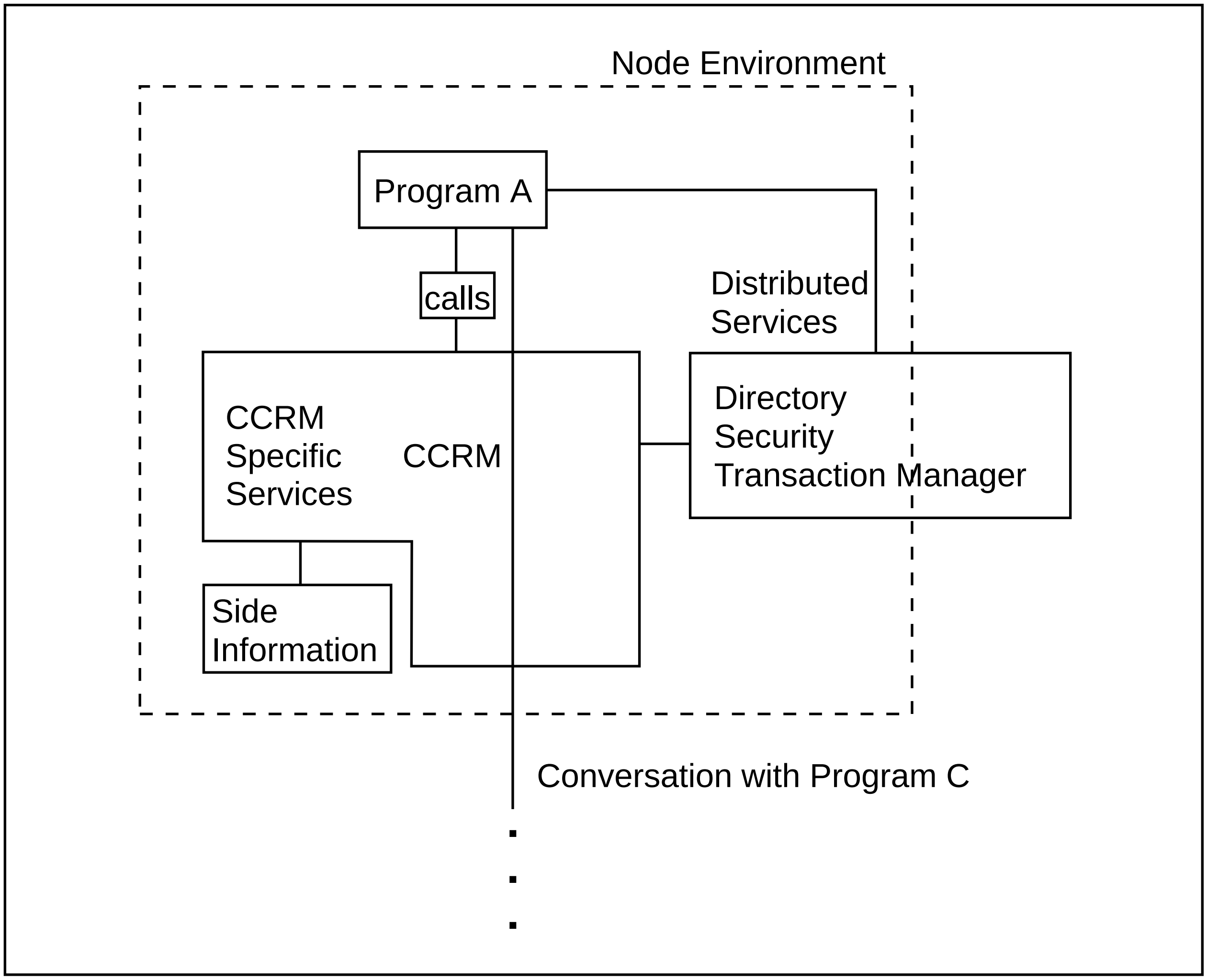


Figure 2. Operating Environment of CPI-C Program

Figure 2 also shows additional elements used by the CCRM or program A such as:

- CCRM-specific services, which include:
  - Side information access
  - Program startup
  - Program termination
  - Context support for programs with multiple partners
- Open Blueprint Distributed System Services, which include:
  - Directory
  - Security
  - Transaction Manager

CCRM-Specific Services

CCRM-specific services are the utility functions within the local CCRM that are used to support CPI-C. These functions are not related to the actual sending and receiving of CPI-C data, and specific implementations differ from product to product. CCRM specific services include the following general functions:

- Setting and accessing side information (see Figure 2)

This function is required to set up the initial values of the side information and to allow subsequent modification.

Side information is local configuration information that is used to identify a partner program without interacting with the distributed directory.

- Program startup processing

  A program is started either by receipt of notification that the remote program has issued a CPI-C Allocate call for the conversation or by local (operator) action. In either case, the local operating system sets up the data paths and operating environment required by the program, validates and establishes security parameters under which the program will execute, then allows the program to begin execution. When an Allocate notification is received, the CCRM retrieves the name of the program to be started along with any access security information included in the conversation startup request, then proceeds as if starting a program by local action.

- Program termination processing (both normal and abnormal)

  The program should end all conversations before the end of the program. However, if the program does not end all conversations, the CCRM will abnormally deallocate any open conversations.

- Providing support for programs with multiple partners

  As described in "Multiple Conversations" on page 13, some programs do work on behalf of multiple partners. Each partner is represented by a *context*, which is a collection of logical attributes. The CCRM provides the necessary function and support (through system interfaces) to allow the program to manage different partner contexts.

Distributed Services

CPI-C programs or the CCRM can directly use the following Open Blueprint Distributed System Services:

- Directory

  A program passes a Distinguished Name (DN) or Program Function Identifier (PFID) to CPI-C, and CPI-C accesses the directory to retrieve the destination information for the program. CPI-C supports both universal naming and contextual naming for the DN that is provided to it. Concatenated naming is not used, since it would tie a program installation to a single CCRM instance. Alternatively, a program can use the directory API to directly interface the directory. For example, a program might want to search the directory using application-specific information to determine the correct destination information. When the correct object has been determined, the program can pass the program destination information to CPI-C.

- Security

  The CCRM uses Security Context Management resource manager interfaces to both acquire and validate access security information on behalf of a user. In a distributed system, the CCRM can use the Identification and Authentication resource manager, which provides authentication services to create or validate the access security information.

- Transaction Manager

  The CCRM can use the Transaction Manager resource manager to tie the various programs involved in one (or more) conversations into a single, recoverable logical unit of work.

Send-Receive Modes

CPI-C supports two modes for sending and receiving data on a conversation:

- **Half-duplex**. Only one of the programs has *send control*, which is the right to send data at any time. Send control must be transferred to the other program before that program can send data.
- **Full-duplex**. Both programs can send and receive data at the same time. Thus, both programs have send control.

The send-receive mode on a conversation is determined when the conversation is established using the CPI-C Allocate call.

Program Partners

The programs involved in a conversation are called *partners*. If a CCRM-to-CCRM logical connection exists or can be created between the nodes containing the partner programs, two programs can communicate through the network with a conversation.

The terms local and remote are used to differentiate between different ends of a conversation. When a program is referred to as the *local* program, its partner program is referred to as the *remote* program for that conversation. For example, if Program A is referred to as the local program, Program C is referred to as the remote program. Similarly, if Program C is referred to as the local program, Program A is referred to as the remote program.

Although program partners generally reside in different nodes in a network, the local and remote programs can, in fact, reside in the same node. Two programs communicate with each other the same way, whether they are in the same or different nodes.

Identifying the Partner Program

CPI-C requires a certain amount of destination information, such as the name of the partner program and the name of the CCRM at the partner's node, before it can establish a conversation. Sources for this information include the:

- **Directory**

  The program can use a *directory object*, which is contained in the *distributed directory*. The directory object is identified to CPI-C by either a DN or a PFID. The directory object represents a particular installation of a program and is universally named with a DN. Programs not knowing a DN for their partner program can have the directory searched for objects containing a specific PFID. The PFID is a globally-unique identifier for the function provided by the program.

- **Side Information**

  The program can use destination information contained in local *side information*. The side information is accessed using an 8-byte *symbolic destination name*, which identifies the entry for the partner program. The side information entry can contain either the individual pieces of destination information or the distinguished name of a directory object.

- **Program-Supplied**

  The program can supply the individual destination information components directly.

Destination Information

The destination information that CPI-C needs is:

partner_LU_name The name of the logical unit where the partner program is located.

TP_name The name of the remote program. *TP_name* stands for *transaction program name*. In this paper, the terms *transaction program*, *application program*, and *program* are synonymous, all denoting a program using a CCRM.

mode_name Designates the properties of the logical connection that will be established for the conversation.

Program Calls

CPI-C programs communicate with each other by making program *calls.* These calls establish the *characteristics* of the conversation and exchange data and control information between the programs. An example of a conversation characteristic is *send_receive_mode*, which indicates whether both programs can send data at the same time. Conversation characteristics are described in greater detail in "Conversation Characteristics" on page 11.

The *return_code* output parameter indicates whether a call completed successfully or if an error was detected that caused the call to fail. CPI-C uses additional output parameters on some calls to pass status information to the program. These parameters include the *control_information_received*, *data_received* and *status_received* parameters. Additionally, secondary information can be associated with the return code. Secondary information can be used to further identify the cause of the return code.

Establishing a Conversation

Here is a simple example of how Program A starts a conversation with Program C:

1. Program A issues the Initialize_Conversation call to prepare to start the conversation. The CCRM returns a unique conversation identifier, the *conversation_ID*. Program A will use this *conversation_ID* in all future calls intended for that conversation.
2. Program A issues a Set_Partner_ID call that contains Program C's universal name (that is, a distinguished name).
3. Program A issues an Allocate call to start the conversation.
4. The local CCRM tells the node containing Program C that Program C needs to be started by sending a *conversation startup request* to the partner CCRM. The conversation startup request contains the information necessary to start the partner program and establish the conversation.
5. Program C is started and issues the Accept_Conversation call. Program C receives back a unique *conversation_ID* (not necessarily the same as the one provided to Program A). Program C will use its *conversation_ID* in all future calls intended for that conversation.

After issuing their respective Initialize_Conversation and Accept_Conversation calls, both Program A and Program C have a set of default conversation characteristics for the conversation.

Conversation Characteristics

CPI-C maintains a set of characteristics for each conversation used by a program (*conversation characteristics*). Conversation characteristics are established for each program on a per-conversation basis, and the initial values assigned to the characteristics depend on the program's role in starting the conversation.

Modifying and Viewing Characteristics

In the example described in the “Establishing a Conversation” on page 10 section above, the programs used the initial set of program characteristics that are provided by CPI-C as defaults. However, CPI-C provides calls that allow a program to modify and view the conversation characteristics for a particular conversation.

Note: CPI-C maintains conversation characteristics on a per-conversation basis. Changes to a characteristic will affect only the conversation indicated by the *conversation_ID*. Changes made to a characteristic do not affect future default values.

Automatic Conversion of Characteristics

Some conversation characteristics affect only the function of the local program; the remote program is not aware of their settings. An example of this kind of conversation characteristic is *receive_type*. Other conversation characteristics, however, are transmitted to the remote program or CCRM and, thus, affect both ends of the conversation. For example, the local CCRM transmits the *TP_name* characteristic to the remote CCRM as part of conversation startup.

When an APPC CCRM is used, these conversation characteristics are encoded as EBCDIC characters for transmission. For this reason, CPI-C automatically converts these characteristics to EBCDIC when they are used as parameters on CPI-C calls on non-EBCDIC systems. This means that programmers can use the native encoding of the local system when specifying these characteristics on CPI-C Set calls. Likewise, when these characteristics are returned by CPI-C Extract calls, the characteristics are represented in the local system's native encoding.

Data Conversion

Program-to-program communication typically involves a variety of computer systems and languages. Data conversion support allows application programs to overcome the differences in data representations from different environments. Programmers may use the native encodings of the local system for data records exchanges with partners on systems with different encodings.

Data Buffering and Transmission

If a program uses the initial set of conversation characteristics, data is not automatically sent to the remote program after a Send_Data has been issued, except when the send buffer at the local system overflows. The startup of the conversation and subsequent data flow can occur anytime after the CPI-C Allocate call, because the system stores the data in internal buffers and groups transmissions together for efficiency.

A program can exercise explicit control over data transmission by using one of the calls that cause the immediate transmission of the buffered data.

Program Flow—States and Transitions

A program that is written to use CPI-C is written with the remote program in mind. The local program issues a CPI-C call for a particular conversation knowing that, in response, the remote program will issue another CPI-C call (or its equivalent) for that same conversation. To explain this two-sided programming scenario, CPI-C uses the concept of a *conversation state*. The conversation state determines what the next set of actions can be. When a conversation leaves a state, it makes a *transition* from that state to another.

A CPI-C conversation can be in one of the following states:

State **Description**

Reset There is no conversation for this *conversation_ID*.

Initialize Initialize_Conversation has completed successfully and a *conversation_ID* has been assigned.

Send The program is able to send data on this conversation. This state is applicable only for half-duplex conversations.

Receive The program is able to receive data on this conversation. This state is applicable only for half-duplex conversations.

Send-Pending The program has received both send control and data on the same Receive call. This state is applicable only for half-duplex conversations.

Confirm A confirmation request has been received on this conversation. After responding with Confirmed, the local program's end of the conversation enters Receive state. This state is applicable only for half-duplex conversations.

Confirm-Send A confirmation request and send control have both been received on this conversation. After responding with Confirmed, the local program's end of the conversation enters Send state. This state is applicable only for half-duplex conversations.

Confirm-Deallocate A confirmation request and deallocation notification have both been received on this conversation.

Initialize-Incoming Initialize_For_Incoming has completed successfully and a *conversation_ID* has been assigned for this conversation. The program can accept an incoming conversation by issuing Accept_Incoming on this conversation.

Send-Receive The program can send and receive data on this conversation. This state is applicable only for full-duplex conversations.

Send-Only The program can only send data on this conversation. This state is applicable only for full-duplex conversations.

Receive-Only The program can only receive data on this conversation. This state is applicable only for full-duplex conversations.

A conversation starts out in Reset state and moves into other states, depending on the calls made by the conversation program and the information received from the remote program. The current state of a conversation determines what calls the program can make.

Because there are two programs for each conversation (one at each end), the state of the conversation as seen by each program can be different. The state of the conversation depends on which end of the conversation is being discussed. Consider a half-duplex conversation where Program A is sending data to

Program C. Program A's end of the conversation is in Send state, but Program C's end is in Receive state.

Note: CPI-C keeps track of a conversation's current state, as should the program. If a program issues a CPI-C call for a conversation that is not in a valid state for the call, CPI-C will detect this error and return a *return_code* value of CM_PROGRAM_STATE_CHECK.

The following additional states exist for conversations that use a transaction manager:

- Defer-Receive (for half-duplex conversations only)
- Defer-Deallocate
- Prepared
- Sync-Point
- Sync-Point-Send (for half-duplex conversations only)
- Sync-Point-Deallocate

“Support for Transaction Managers” on page 21 discusses synchronization point processing and describes these additional states.

Multiple Conversations

In the example described in “Establishing a Conversation” on page 10, Program A established a single conversation with a single partner. CPI-C allows a program to communicate with multiple partners using multiple, concurrent conversations, such as:

- **Outbound Conversations**. A program initiates more than one conversation.

- **Inbound Conversations** A program accepts more than one conversation.

Specific combinations of outbound and inbound conversations are determined by application design. The following sections discuss in greater detail the concepts required for multiple conversations.

Partner Program Names

After a program issues Initialize_Conversation to establish its conversation characteristics, a name for its partner program (the TP_Name) is established. This name is transmitted to the remote system in the conversation startup request after the program issues the Allocate call.

At the remote system, the partner program can be started in one of two ways:

- Receipt of a conversation startup request
- Local action

In the first case, the CCRM starts the program named in the conversation startup request. However, if a program is started locally, the program must notify the CCRM of its ability to accept conversations for a given name. The program associates a name with itself by issuing the Specify_Local_TP_Name call. The program can release a name from association with itself by issuing the Release_Local_TP_Name call.

Multiple Outbound Conversations

Figure 3 on page 14 shows Program A establishing conversations with two partners. For example, a program might need to request data from multiple data bases on different nodes to answer a particular query. The conversation with Program B is initialized with an Initialize_Conversation (CMINIT) call that returns a *conversation_ID* parameter of X. The conversation with Program C is initialized with an Initialize_Conversation call that returns a *conversation_ID* parameter of Y. When Program A issues subsequent calls with a *conversation_ID* of X, CPI-C will know these calls apply to the conversation with

Program B. Similarly, when Program A issues subsequent calls with a *conversation_ID* of Y, CPI-C will know these calls apply to the conversation with Program C.

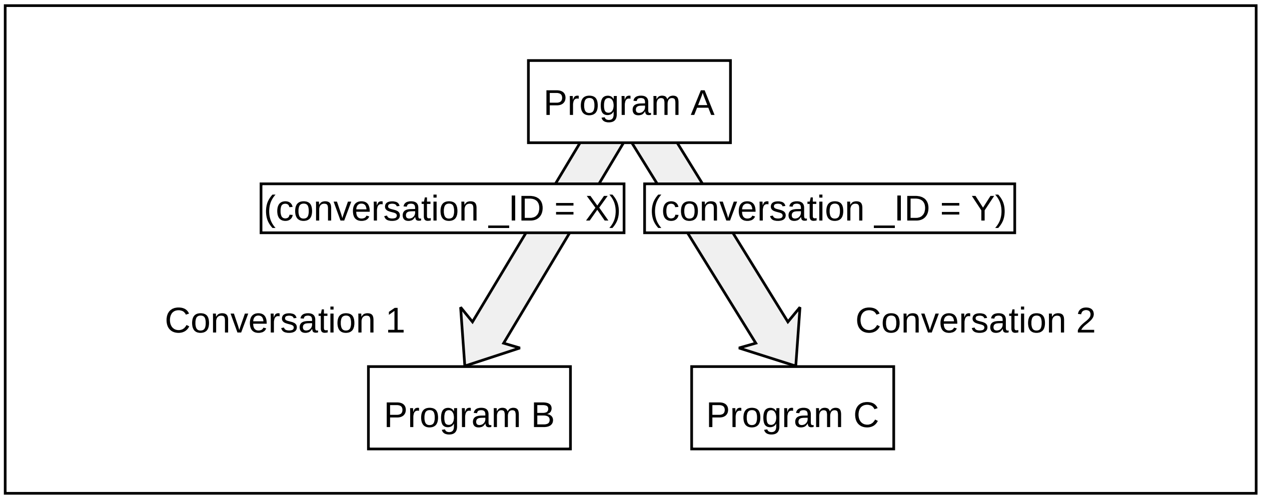


Figure 3. A Program Using Two Outbound Conversations

Multiple Inbound Conversations

Programs can accept more than one inbound conversation. For example, a server could accept conversations from multiple partners in order to work on the request from one partner while waiting for a second partner's request or work to complete.

This type of program is shown in Figure 4, where Programs D and E have both chosen to initiate conversations with the same partner, Program S.

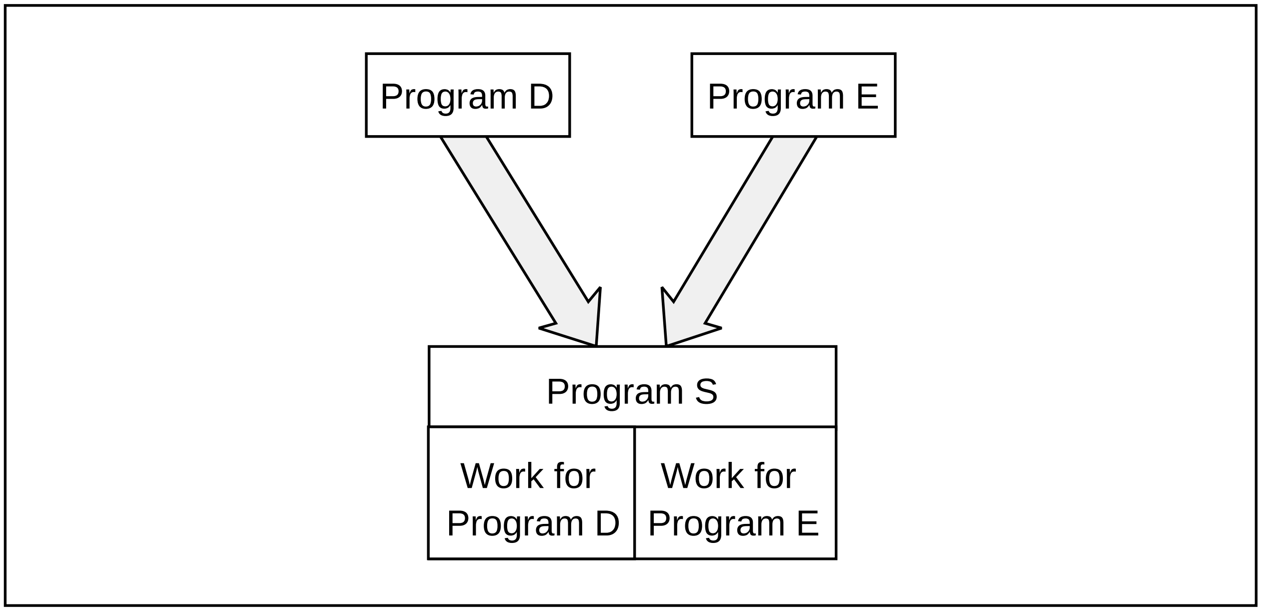


Figure 4. A Program Using Two Inbound Conversations

In the simplest case, Program S can accept the two conversations by issuing the CPI-C Accept_Conversation call twice. Alternatively, Program S can use the CPI-C Initialize_For_Incoming and Accept_Incoming calls.

Contexts and Context Management

CPI-C provides support for programs that perform work on behalf of multiple partners, such as Program S in the previous example. Each time a program accepts an incoming conversation, a new context is created. The context is identified by a *context identifier*, which is used to group logical attributes for the work to be done on behalf of the partner program.

CPI-C maintains one or more contexts for a program running within the node. For each program, there is one distinguished context, the *current context*, within which work is currently being done. A program can manage different contexts by making calls to the local operating system's environment state support to:

- Create a new context
- Terminate a context
- Set the current context
- Retrieve the context identifier of the current context

Note: The discussion of context throughout this and following sections assumes that a context is maintained on a per-program basis. However, in a system that supports multi-threaded programs, the context can be maintained on a per-thread basis.

Relationship between Context and Conversation

Each conversation is assigned to a context when allocated or accepted.

- An outgoing conversation is assigned to the current context of the program when the program issues the CPI-C Allocate call.
- An incoming conversation is assigned to the new context that is created when the program accepts the incoming conversation with either the CPI-C Accept_Conversation or the CPI-C Accept_Incoming call.

A program can retrieve the context identifier for a conversation's context by issuing the Extract_Conversation_Context call.

The program's current context is set by the CCRM to the newly-created context when an Accept_Conversation or Accept_Incoming call completes successfully with the *return_code* set to CM_OK.

Inbound and Outbound Conversations

A program that accepts incoming conversations from multiple program partners must ensure that work is done within the correct context. In the expanded server example shown in Figure 5 below, Program S accepts two conversations, one each from Programs D and E. Two new contexts are created, one for the work done for Program D and one for the work done for Program E. Program S can retrieve the context identifier for each conversation by issuing the Extract_Conversation_Context call twice.

In a different scenario, Program S is doing work for Program E and receives a request from Program D. The request is for information that Program S does not have. In this case, Program S allocates a conversation to a third partner, Program F, to answer Program D's request.

Before allocating the conversation to Program F, Program S must first ensure that it is using the correct context by setting its current context to that for Program D. This causes the outgoing conversation (to Program F) to be established using information from the context for Program D (for example, the security parameters).

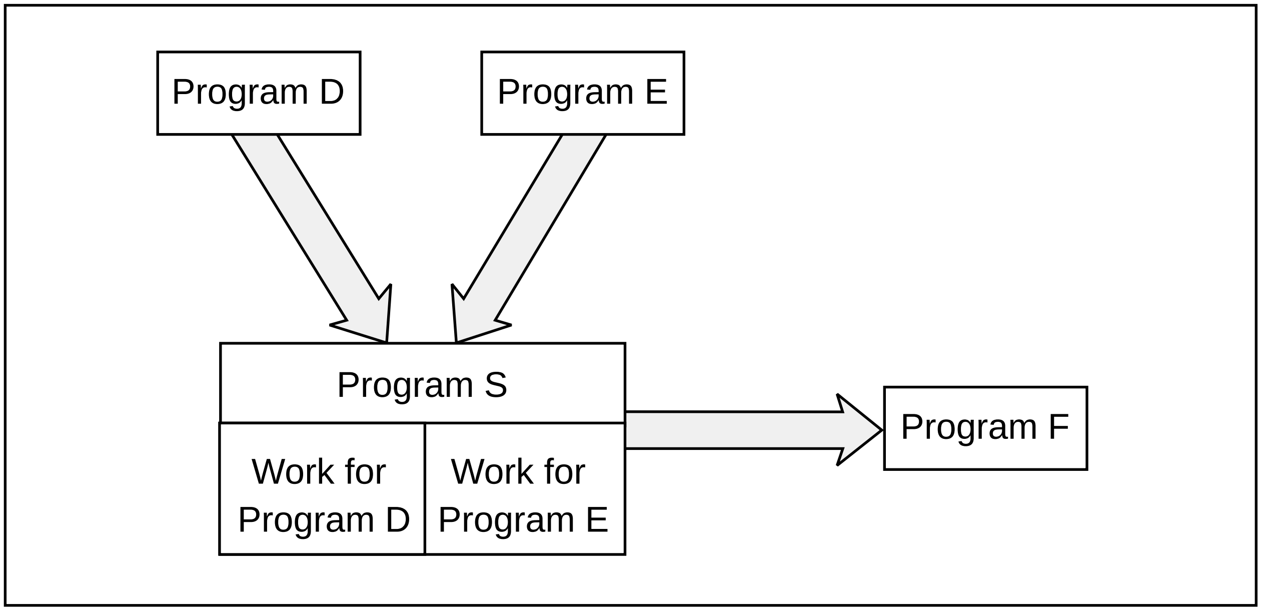


Figure 5. Program with both Inbound and Outbound Conversations

Concurrent Operations

CPI-C provides for concurrent call operations (multiple call operations in progress simultaneously) on a conversation by grouping calls in logical associations or *conversation queues*. Calls associated with one queue are processed independently of calls associated with other queues or no queue.

A program can initiate concurrent operations by using multiple program threads on systems with multi-threading support. Alternatively, a program can use queue-level, non-blocking support to regain control when a call operation on a queue cannot complete immediately. The call operation remains in progress. The program can issue a call associated with another queue or perform other processing.

Only one call operation is allowed to be in progress on a given conversation queue at a time. If a program issues a call associated with a queue that has a previous call operation still in progress, the more recent call returns a *return_code* set to CM_OPERATION_NOT_ACCEPTED.

Non-Blocking Operations

CPI-C supports two processing modes for its calls:

Blocking The call operation completes before control is returned to the program. If the call operation is unable to complete immediately, it blocks, and the program is forced to wait until the call operation finishes. While waiting, the program is unable to perform other processing or communicate with any of its other partners.

Non-blocking If possible, the call operation completes immediately and control is returned to the program. However, if while processing the call CPI-C determines that the call operation cannot complete immediately, control is returned to the program even though the call operation has not completed. The call operation remains in progress, and completion of the call operation occurs at a later time.

Note: This section describes non-blocking operations for a single-threaded program, but similar considerations apply to a program issuing CPI-C calls on multiple threads. Specifically, only the thread that issues a call is blocked if the call is processed in blocking mode and cannot complete immediately.

When the program uses non-blocking support, control is returned to the calling thread if the call operation cannot complete immediately. That thread can then perform other processing, including issuing calls on the same conversation.

When the non-blocking processing mode applies to a call and the call operation cannot complete immediately, CPI-C returns control to the program with a return code of CM_OPERATION_INCOMPLETE. The call operation remains in progress as an *outstanding operation*, and the program is allowed to perform other processing.

CPI-C provides two levels of support for programs using the non-blocking processing mode: *conversation* and *queue*. These modes are described in the following sections. Until a program chooses a non-blocking level for a conversation, all calls on the conversation are processed in blocking mode.

Conversation-Level Non-Blocking

Conversation-level non-blocking allows only one outstanding operation on a conversation at a time. The program chooses conversation-level non-blocking by issuing the Set_Processing_Mode (CMSPM) call to set the *processing_mode* conversation characteristic.

The program issues the Wait_For_Conversation call to determine when outstanding operations are completed and to retrieve the return code for that operation. CPI-C keeps track of all conversations that are using conversation-level non-blocking and that have an outstanding operation. CPI-C responds to a subsequent Wait_For_Conversation call with the conversation identifier of one of those conversations when the operation on the conversation completes.

Queue-Level Non-Blocking

In contrast to conversation-level non-blocking, queue-level non-blocking allows more than one outstanding operation per conversation. CPI-C allows programs using queue-level non-blocking to have one outstanding operation per queue simultaneously.

With queue-level non-blocking, the processing mode is set on a per-queue basis. The program chooses queue-level non-blocking by issuing the Set_Queue_Processing_Mode or Set_Queue_Callback_Function call to set the queue processing mode for a specified queue. Until the program sets the processing mode for a queue, all calls associated with that queue are processed in blocking mode. For example, a full-duplex program can use blocking for sending Data calls, while using non-blocking for CPI-C Receive calls. CPI-C calls that are not associated with any queue are processed in blocking mode and are always completed before control is returned to the program.

Working with Wait Facility: When using the Set_Queue_Processing_Mode call, the program manages multiple outstanding operations with *outstanding-operation identifiers* (OOIDs). CPI-C creates and maintains a unique OOID for each queue. Additionally, a program can choose to associate a *user field* with an outstanding operation. The *user field* is provided as an aid to programming, and might be used, for example, to contain the address of a data structure with return parameters for an outstanding operation.

The program issues the Wait_For_Completion call to wait for the outstanding operations to complete and obtain the corresponding OOID and user field.

Using A Callback Function: An alternate use of queue-level non-blocking is to establish a *callback function* and a user field for the conversation queue by using the CPI-C Set_Queue_Callback_Function call. When an outstanding operation completes, the program is interrupted and the callback function is called. When the callback function returns, the program continues from where it was interrupted.

Distributed Directory

The local directory interface handles the communications and information flows required to retrieve the requested information from the directory. Information is stored in the directory by placing it in a directory object. Directory objects can contain many different pieces of information.

Figure 6 shows Program A interacting directly with a local directory interface. When Program A provides a name to the directory client (1), the directory client service accesses the distributed directory (2 and 3). The retrieved directory object is then returned to the program (4).

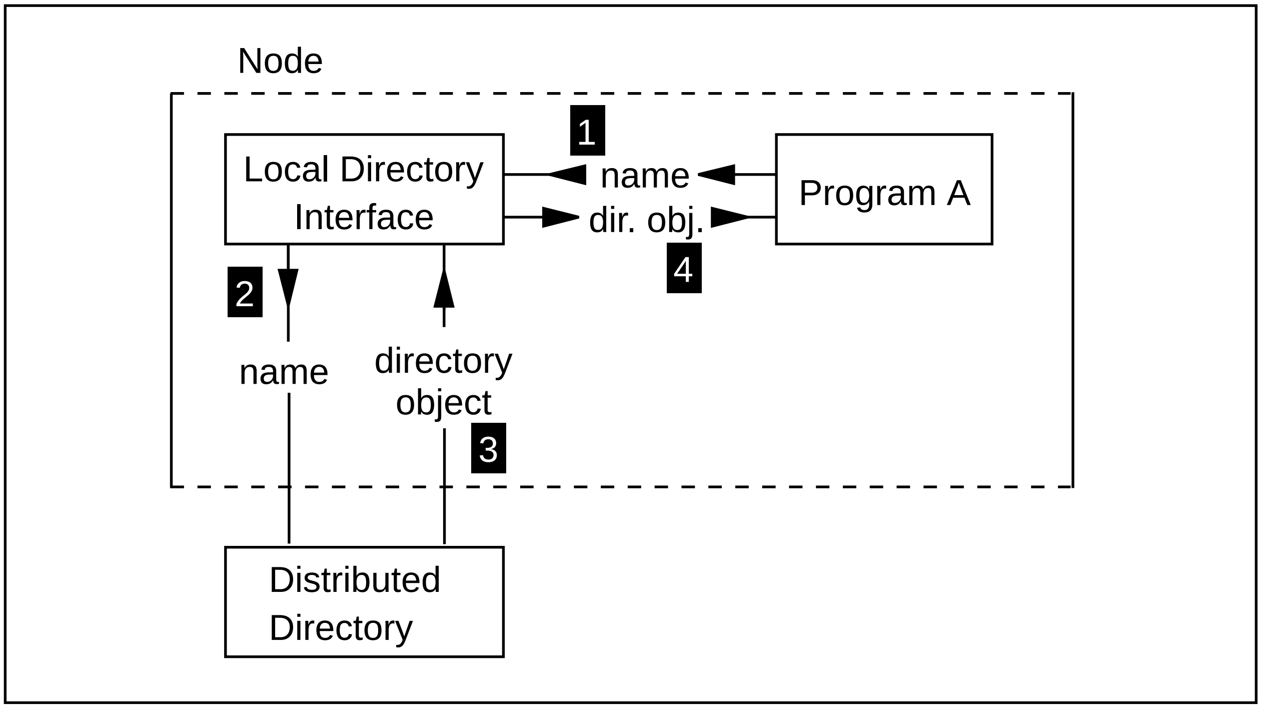


Figure 6. Program Interaction with a Distributed Directory

CPI-C Directory Object

A CPI-C directory object that contains the destination information for a single installation of a partner program is referred to as a *program installation object*. A program installation object includes the following information:

- **Program Function Identifier**. A program function identifier (PFID) uniquely identifies the function provided by a program. The PFID allows programs installed on multiple systems, with different DNs, to be recognized as providing the same function. For example, multiple installations of a distributed mail application can all have the same PFID.
- **Program Binding**. The program binding contains information required by a partner program to establish a conversation with the program. Because multiple CCRM instances can be used to reach

the same program installation, there can be more than one program binding in a single directory object. Each program binding contains the following information:

Destination Information TP_name, partner_lu_name, and mode_name

partner_principal_name Identifies the principal name used by the remote CCRM for authentication of conversation startup requests

required_user_name_type Identifies the type of user name required to access the partner program

Using the Distributed Directory

Figure 7 below illustrates two ways that a CPI-C program might use destination information stored in a distributed directory:

- Program A accesses the distributed directory directly with a DN to retrieve a program installation object (1 and 2). The program installation object contains a program binding as one of its pieces of information. Program A passes the program binding to CPI-C (3) using the Set_Partner_ID call with the *partner_ID_type* parameter set to CM_PROGRAM_BINDING. CPI-C then uses the program binding information to allocate the conversation.
- Instead of accessing the directory itself, Program A passes a DN for a program installation object to CPI-C (3) using the Set_Partner_ID call with a *partner_ID_type* parameter set to CM_DISTINGUISHED_NAME. CPI-C uses the DN to access the distributed directory and retrieve the program installation object (4). CPI-C then uses the program binding information from the object to allocate the conversation.

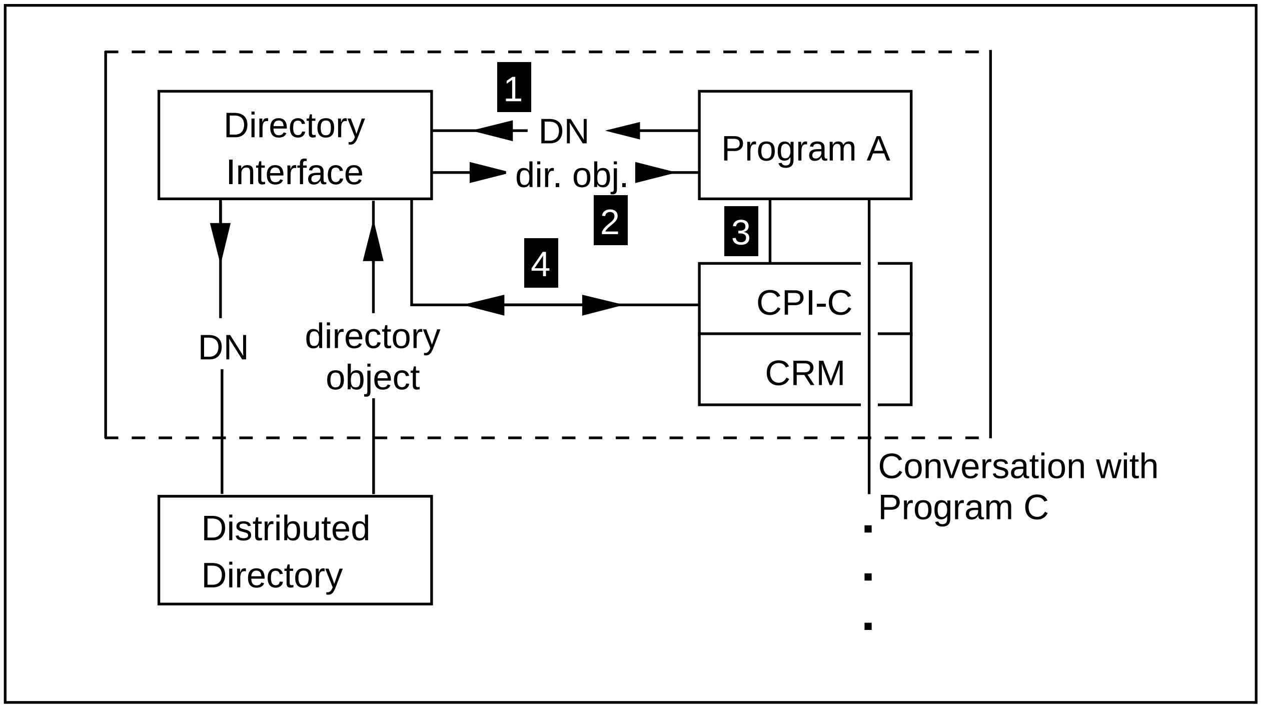


Figure 7. Program Interaction with CPI-C and a Distributed Directory

Both of these examples require the program to provide a DN directly to either the directory service or CPI-C. There are, however, several other ways a program can access information that is contained in the distributed directory.

- The program passes a PFID to CPI-C. CPI-C uses the PFID to search the distributed directory and retrieve a program installation object that provides the appropriate function. CPI-C then uses the program-binding information from the object to allocate the conversation.
- The program provides a symbolic destination name on the Initialize_Conversation call that corresponds to a side information entry containing a DN. CPI-C then uses the DN to access the distributed directory and retrieve the program binding. After the conversation is allocated, the program can determine which program binding is used.
- The program accesses the distributed directory and locates the appropriate directory object (and program binding) using something other than a DN. For example, the program might search the directory using application-specific information to locate the correct program installation object. When the directory object is located, either the program binding from the directory object or the name of the object can be passed to CPI-C.

Conversation Security

Many systems control access to system resources using security parameters that are associated with a request for access to those resources. In particular, a CCRM working in conjunction with local and distributed security services can control access to its programs and conversation resources using access security information carried in the conversation startup request.

The conversation startup request contains one of the following forms of access security information:

- Credentials generated by a distributed security service for the user on whose behalf access to the remote program is requested
- The user ID of the user on whose behalf access to the remote program is requested
- The user ID and a password for the user on whose behalf access to the remote program is requested
- No access security information

When a program is started as a result of an incoming conversation startup request or a program that is already started accepts an incoming conversation, the CCRM uses the access security information to validate the user's access to the program and establish the security parameters for the resulting context.

Distributed Security

Distributed security allows a user or system to be defined at a trusted authentication server using *principal names* rather than user IDs. For example, a user or program accessing three different systems might require three separate sets of user IDs and passwords, one for each system. Using distributed security, a user or system could use a single principal name and password to access all three systems.

Figure 8 below shows how a distributed security service works. In this example, the user has already signed on with a principal name and has been authenticated to the local security service interface. After this initial sign-on, the user is not required to provide any additional security information. When the user executes a program that requests a conversation from the CCRM (**1**), the CCRM communicates with the local security service interface (**2**) and retrieves the security information to be sent with the conversation startup request. The local security service communicates with the authentication server (**3**) to determine this information, also referred to as *credentials*, and returns it to the CCRM (**4**). The CCRM then sends the credentials to the partner CCRM in the conversation startup request (**5**). When the partner CCRM receives the conversation startup request, it uses the local security service interface to validate the credentials (**6** and **7**).

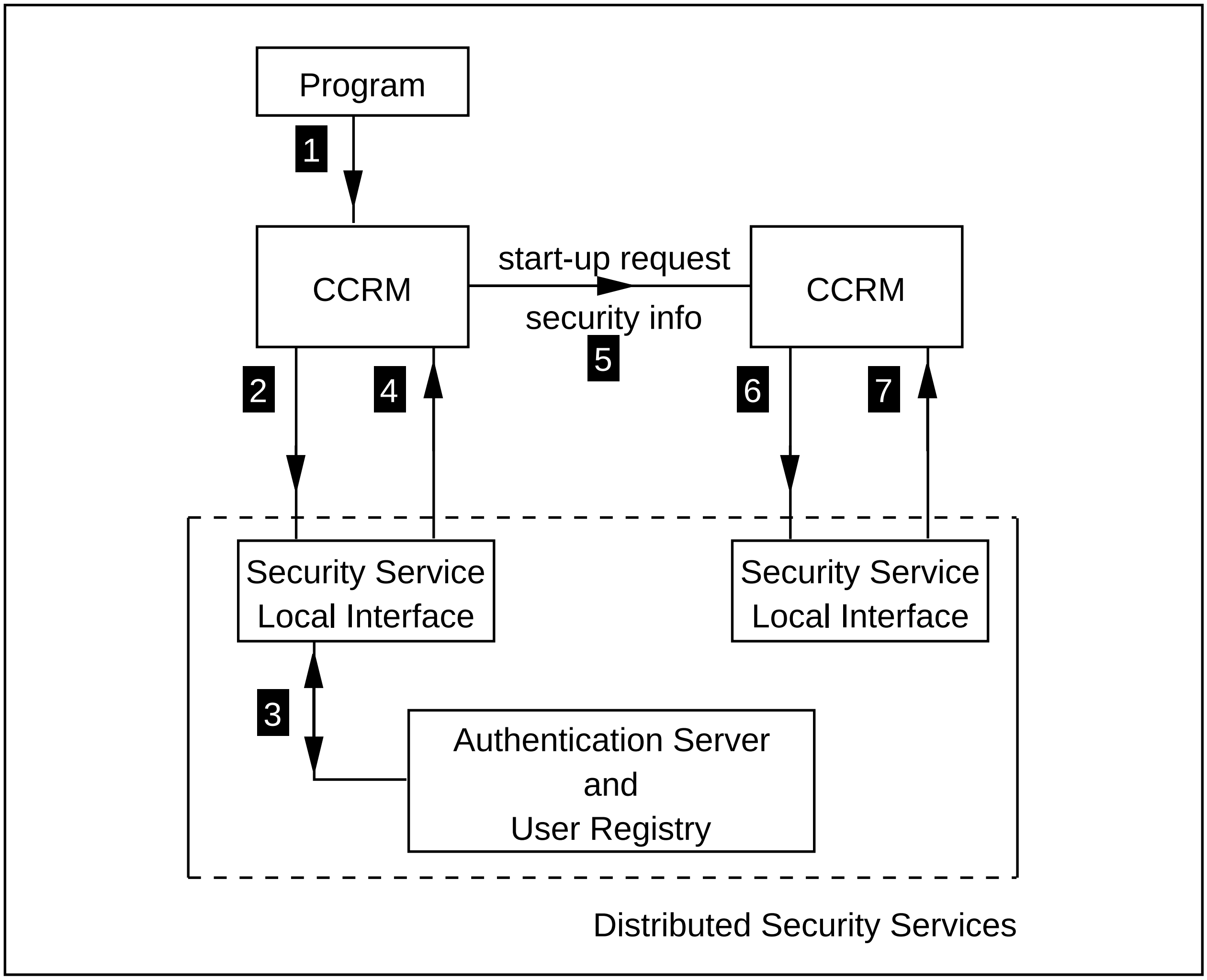

Program
1
CCRM
start-up request
security info
5
CCRM
2
4
6
7
Security Service Local Interface
Security Service Local Interface
3
Authentication Server and User Registry
Distributed Security Services

Figure 8. CCRM Interaction with Distributed Security Service

Support for Transaction Managers

A *transaction manager* provides access to services and facilities that use two-phase commit protocols to coordinate changes to distributed resources. While CPI-C sync point functions can use other transaction managers, this paper describes how the CCRM works with the Transaction Manager resource manager using the X/Open TX resource recovery interfaces.

A CPI-C conversation uses a transaction manager only if its *sync_level* characteristic is set to CM_SYNC_POINT or CM_SYNC_POINT_NO_CONFIRM. This kind of conversation is called a *protected conversation*.

Coordination with Transaction Managers

A program communicates with a transaction manager by establishing *synchronization points* (or *sync points*) in the program logic. A sync point is a reference point that is used during transaction processing to establish a state to which resources can be restored if a failure occurs. The program uses the transaction manager's commit call (for example, tx_commit) to establish a new sync point or a transaction manager's backout call (for example, tx_rollback) to return to a previous sync point. The processing and the changes to resources that occur between one sync point and the next are collectively referred to as a *logical unit of work*.

The transaction manager coordinates commit and backout processing among all the protected resources involved in the logical unit of work.

The commitment or backout of protected resources is done on a context basis. Only those changes to protected resources, including protected conversations, that belong to the program's current context when the tx_commit or tx_rollback call is issued are committed or backed out. Therefore, prior to issuing the tx_commit or tx_rollback call, the program must ensure that the current context is the context for which the commit or backout is intended.

For example, Program S in Figure 4 on page 14 has two distinct contexts, one for Program D and one for Program E. If Program S decides to commit or back out the work that it has done for Program D, it must set the current context to ensure that only the resources associated with Program D will be affected. For more information on contexts, see “Contexts and Context Management” on page 15.

Take-Commit and Take-Backout Notifications

When a program issues a Prepare, tx_commit, or tx_rollback call, the CCRM cooperates with the transaction manager by passing synchronization information to the program's conversation partners. This synchronization information consists of take-commit and take-backout notifications.

When the program issues a Prepare or tx_commit call, CPI-C returns a take-commit notification to the partner program in the *status_received* parameter for a CPI-C Receive call issued by the partner. The sequence of CPI-C calls issued before the Prepare or tx_commit call determines the value of the take-commit notification that is returned to the partner program. In addition to requesting that the partner program establish a sync point, the take-commit notification also contains conversation state transition information.

When the program issues a tx_rollback call, or when a system failure or a problem with a protected resource causes the transaction manager to initiate a backout operation, CPI-C returns a take-backout notification to the partner program.

The Backout-Required Condition

Upon receipt of a take-backout notification on a protected conversation, the conversation's context is placed in the Backout-Required condition. This condition is not a conversation state, because it applies to all of the program's protected resources for that context, possibly including multiple conversations.

A context can be placed in the Backout-Required condition in one of the following ways:

- When CPI-C returns a take-backout notification
- When the program issues a Cancel_Conversation call
- When the program issues a Deallocate call with *deallocate_type* set to CM_DEALLOCATE_ABEND
- When the program issues a Send_Data call with *send_type* set to CM_SEND_AND_DEALLOCATE and *deallocate_type* set to CM_DEALLOCATE_ABEND

When a context is placed in the Backout-Required condition, the program should issue a tx_rollback call. Until it issues a tx_rollback call, the program will be unable to successfully issue most CPI-C calls for any protected conversations within that context.

Additional Conversation States

In addition to the states described in “Program Flow—States and Transitions” on page 12, a protected conversation can also be in one of the following states:

State **Description**

Defer-Receive The local program's end of the conversation will enter the Receive state after a synchronization call completes successfully. The synchronization call can be a tx_commit call or a CPI-C Flush or Confirm call.

Defer-Deallocate The local program requested that the conversation be deallocated after a commit operation has completed.

Prepared The local program issued a Prepare call to request that the remote program prepare its resources for commitment.

Sync-Point The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT or CM_TAKE_COMMIT_DATA_OK.

Sync-Point-Send The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT_SEND or CM_TAKE_COMMIT_SEND_DATA_OK.

Sync-Point-Deallocate The local program issued a Receive call and was given a *return_code* of CM_OK and a *status_received* of CM_TAKE_COMMIT_DEALLOCATE or CM_TAKE_COMMIT_DEALLOC_DATA_OK.

Alternative Conversational Semantics

As an alternative to using CPI-C, minimal conversational semantics can be obtained by using the Sockets/Winsock interface to TCP/IP. For a more complete description of the Sockets/Winsock interface, see the *Open Blueprint Network Services* component description paper.

Additional Information Sources

Online Information Sources

Online information on CPI-C, APPC, and APPN is available through CompuServe, the OS/2 Bulletin Board Service, and the Internet. Up-to-date information on CPI-C, access sample programs and tools, ask questions, and provide feedback on CPI-C and related IBM products is available from any of these online sources.

CompuServe: IBM maintains the APPC/APPN Forum (GO APPC) on CompuServe. More than a dozen question-and-answer sections exist as well as hundreds of sample programs, utilities, and technical papers.

OS/2 Bulletin Board Service: The OS/2 BBS APPC FORUM is available on the IBM Global Network (IGN) and the IBMLink TalkLink facility. In the US, the phone number is 1-800-547-1283; in Canada, 1-800-465-7999 (ext 228). Elsewhere, contact your local IBM representative.

Internet: A wide variety of announcements, utilities, and sample programs are available from the Internet anonymous FTP site: *ftp://networking.raleigh.ibm.com/pub/appc_appn*. The CIW home page is at url: *http://www.raleigh.ibm.com/app/aiwconf/aiwciw.htm*. IBM also monitors the bit.listserv.appc-l USENET newsgroup.

[1] The CPI-C Implementers Workshop (CIW) is a forum of CPI-C architects, implementers and users. The CIW's purpose is to define enhancements to CPI-C and promote the implementation and use of CPI-C.

[2] Communications resource managers can provide many functions in a network. In this paper, the term *conversational communication resource manager* refers to resource managers that provide conversation services to programs. Other CRMs provide for remote procedure calls or message-queuing services.

Appendix A. Bibliography

- *Open Blueprint Technical Overview*, GC23-3808
- *Systems Network Architecture Technical Overview*, GC30-3073
- *Distributed Transaction Processing: The XCPI-C Specification, Version 2*, X/Open CAE Specification C419. December, 1995. ISBN 1-85912-135-7
- *CPI Communications: CPI-C Specification Version 2.1*, SC31-6180

Appendix B. Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, may functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

APPN
IBM
IBMLink
Open Blueprint
OS/2
Talklink

The following terms are trademarks of other companies:

CompuServe	CompuServe, Incorporated and H&R Block, Incorporated
DCE	The Open Software Foundation
X/Open	X/Open Company Limited in the U.K. and other countries

Appendix C. Communicating Your Comments to IBM

If you especially like or dislike anything about this paper, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply. Feel free to comment on specific error or omissions, accuracy, organization, subject matter, or completeness of this paper.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

- If you prefer to send comments by FAX, use this number:

  United States and Canada: 1-800-227-5088.

- If you prefer to send comments electronically, use one of these ID's:
  - Internet: **USIB2HPD@VNET.IBM.COM**
  - IBM Mail Exchange: **USIB2HPD at IBMMAIL**
  - IBMLink: **CIBMORCF at RALVM13**

Make sure to include the following in your note:

- Title of this paper
- Page number or topic to which your comment applies

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

GC23-3914-01