NISTIR 5657

An Introduction to the P1003.1g and CPI-C Network Application Programming Interfaces

Karen Olsen

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

NIST

NISTIR 5657

An Introduction to the P1003.1g and CPI-C Network Application Programming Interfaces

Karen Olsen

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

May 1995

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown, Secretary

TECHNOLOGY ADMINISTRATION
Mary L. Good, Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS
AND TECHNOLOGY
Arati Prabhakar, Director

Abstract

Numerous network application programming interfaces (APIs) have been developed to assist programmers in developing distributed applications. Both IEEE P1003.1g and Common Programming Interface for Communications (CPI-C) are examples of network APIs. This report provides an overview of the P1003.1g and CPI-C specifications. An overview of the basic functionality of P1003.1g and CPI-C calls is given, along with a simple programming example for each API. This report does not contain sufficient detail needed to develop distributed applications using these network APIs, however, a list of references is provided.

Contents

List of Figures

1 Introduction

Over the past decade, as computer networks have been designed and implemented, it has become desirable to re-engineer local applications into *distributed applications*. Many *application programming interfaces* (APIs) have been developed to assist the programmer in developing applications. Network APIs have been developed to assist the programmer in developing applications that can be ported across a disparate range of operating systems. With the recent move towards open systems, networks are becoming more diverse in the variety of equipment from which the network is implemented. As heterogeneous computing environments have evolved, it has become desirable for distributed applications to be capable of running over a variety of platforms and protocols.

Over the past several years, the range of APIs being developed for data communication services has broadened. This paper[1] introduces the P1003.1g (formerly known as P1003.12) and Common Programming Interface-Communications (CPI-C) specifications. An overview of each of these specifications is provided.

The Introduction presents the underlying concepts behind the P1003.1g and CPI-C interfaces. Sections 2 and 3 provide an overview of the P1003.1g and CPI-C specifications. Sections and 4 and 5 take a closer look at the P1003.1g Detailed Network Interface (DNI). Section 3 describes CPI-C 2.0, from which the latest IBM and X/Open versions of CPI-C are based. Appendices A, B, and C provide programming examples using the XTI, Sockets and CPI-C interfaces [11].

1.1 Distributed Applications

Before computer networks became popular, applications were developed and run locally (e.g. on a mainframe). Applications, that previously were only run locally, can be distributed to take advantage of remote resources. *Interprocess communication* (IPC) involves the transfer of data between processes. IPC can occur between processes on the same system as well as between processes on different systems. Most facilities for interprocess communication are designed such that data is transferred between one or more *end-points* within each process. This allows a process to use more than one interprocess communications channel per process.

A distributed application is a program that is partitioned among multiple processes, and possibly spread across multiple machines. Common characteristics among distributed applications include local components, communications components, and remote components. A local component of a distributed application consists of the end-user, programmatic interfaces to the application, functions that access local input/output resources (e.g. a disk), and access to computing resources of the local system. A communications component consists of the entity that provides distributed communications capabilities to the distributed appli-

[1]Because of the nature of this report, it is necessary to mention vendors and commercial products. The presence or absence of a particular trade name product does not imply criticism or endorsement by the National Institute of Standards and Technology, nor does it imply that the products identified are necessarily the best available

cation. The communications component provides the ability to transfer data between the local systems and one or more remote systems. A remote component consists of functions that access input/output resources and computing resources of a remote system.

Examples of distributed applications are file transfer applications and applications which facilitate the exchange of mail. A goal of distributed applications is that the use of multiple autonomous systems remain transparent to the user.

1.2 Client/Server Model

The *client/server model* of communications is a model of interaction in a distributed system in which one process contacts another process to request a service. The process requesting a service is called a *client.* The process satisfying client requests is called a *server.* A server can also be a client. In such cases, the process is referred to as a *client/server.* Quite frequently, a system which provides services to other systems is called a server node. Since this report deals with interprocess communication among multiple processes on multiple systems, some of which will be clients and some of which will be servers or client/servers, the terms client and server will be used to refer to processes rather than nodes.

1.3 Protocols and Layering

The Open System Interconnection (OSI) Reference Model is commonly used for describing network architectures. The idea behind the OSI Reference model is to group functions into layers to make the implementation of protocols more manageable. The OSI Reference Model provides a framework in which standards can be developed for the services and protocols at each layer. Each layer is based upon the previous layer.

OSI, the Transmission Control Protocol/Internet Protocol (TCP/IP) protocol suite (also know as the Internet Protocol Suite), and Systems Network Architecture (SNA) each have their own network architecture. Figure 1 shows one interpretation of how the SNA and TCP/IP architectures compare to the OSI reference model [9]. Note that layers of the SNA and TCP/IP architectures do not map cleanly to the OSI architecture. For example, the Internet Protocol Suite specifies only 5 layers, combining the functions of the OSI application, presentation, and session layers into a single application layer. See references [9], [11], [12] and [13] for more information on these architectures.

Each network architecture tends to have its own terminology and definition of the services provided by each layer. In very general terms, communications can be viewed as consisting of three relatively independent layers: network access layer, transport layer, and application layer. The network access layer is concerned with the exchange of data between a system and the network to which it is attached. The specific software used at the network access layer depends on the type of network to be used. The transport layer is a common layer shared by all applications and is responsible for the reliable exchange of data between different processes in different systems. The *transport provider* or *communications provider* is the set of routines on the host system that provide communication support for a user process. The

OSI		TCP/IP Protocol Suite		SNA	
7	Applications	7	Process/ Application	7	Transaction services
6	Presentations	6		6	Presentation services
5	Session	5	Host-Host	5	Data flow control
4	Transport	4		4	Transmission control
3	Network	3	Internet	3	Path control
2	Data Link	2	Network access	2	Data link control
1	Physical	1		1	Physical control

Figure 1: A Comparison of Communications Architectures

application layer contains the logic needed to support user applications. For each different type of application, a separate module is needed. This report will focus on the services provided by the transport layer. The transport layer provides the basic service of reliable, end-to-end data transfer needed by applications and higher layer protocols.

1.4 Data Transfer

A transport protocol is responsible for transferring data between two communicating endpoints. Data includes both user data and control data. Full-duplex and half-duplex are two modes of data transfer. For half-duplex data transfer, only one of the communicating programs has the right to send data at any time. Send control must be transferred to the other communication endpoint before that endpoint can send data. For full-duplex data transfer, both communicating endpoints can send and receive data simultaneously.

1.5 Type of Service

Two basic types of communications service are possible: *connection-oriented* or *connectionless* service. Connectionless service is commonly used by applications that involve short-term communication, such as request/response interactions. Connection-oriented communication

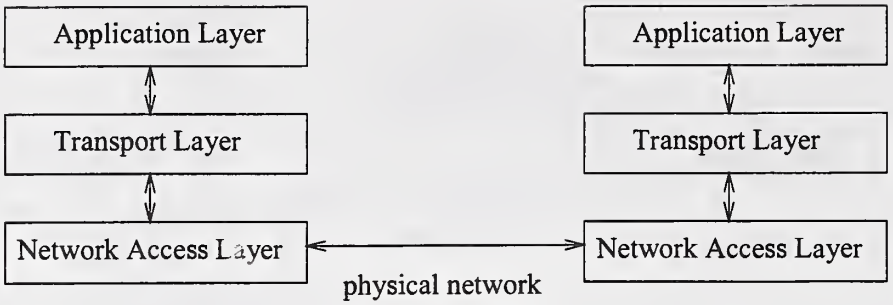

Figure 2: Simplified Communications Architecture

service is commonly used by applications that require relatively long-lived, data stream-oriented interactions.

Connectionless service, also known as datagram service, is message oriented and supports data transfer in self-contained units called datagrams. Since datagrams are transmitted independently, each datagram must contain all the information required for its delivery.

Connection-oriented service requires that two end-points establish a logical connection with each other before communication can take place. There is some overhead associated with establishing the connection. This overhead is more than made up for during the data transfer phases. Some systems have a limit on the number of connections they can support.

A connection-oriented data exchange involves three steps: connection establishment, data transfer, connection termination. Connection establishment is client/server oriented. One process acts as a server that passively waits for incoming connections, while the other process acts as a client that actively initiates a connection..

Connection-oriented service generally implies that service is reliable. Flow control, error control, and sequenced delivery are examples of connection-oriented features. For each connection, it is possible to negotiate the parameters and options that govern the data transfer.

1.6 Blocking v.s. Non-blocking Mode

Blocking mode is a mode of execution in which functions wait for requested operations to complete before returning control to the calling application. Non-blocking mode is a mode of execution in which functions do not wait for protocol events to occur before returning control to the calling application. In non-blocking mode, functions must either complete an operation without blocking, or return an error noting that the operation would have blocked.

I/O (input/output) multiplexing can be done on multiple file descriptors. A technique called asynchronous I/O allows the kernel to notify a process with a signal when a descriptor is ready for I/O. When using a technique called I/O multiplexing, a list is built of descriptors that a user is interested in. A function (e.g. *select()* or *poll()* is then invoked and does not return until one of the descriptors is ready for I/O.

2 IEEE P1003.1g

P1003.1g [1], titled Protocol Independent Interfaces, is being developed as one of the IEEE Portable Operating System Interface (POSIX) projects. The full title of IEEE P1003.1g is Information Technology - Portable Operating System Interface (POSIX) - Part xx: Protocol Independent Interfaces (PII). P1003.1g was previously named POSIX.12.

POSIX P1003.1 defines a standard operating system interface to support application portability at the source code level. The P1003.1g protocol independent interfaces, which are extensions to P1003.1, allow portable applications to communicate independent of the underlying protocols. The P1003.1g interfaces are intended to be used by application developers and systems implementors.

P1003.1g specifies a transport layer API. An application performs process-to-process communication by using P1003.1g to access the underlying Communication Services. Communications Services include network facilities such as protocol stacks (e.g. TCP/IP, OSI) which provide access to the underlying network. The protocol independence provided by P1003.1g insulates the application from the specifics of the underlying protocol stack which provides the communication services.

When the P1003.1g working group began developing its draft, the Sockets Interface and the X/Open Transport Interface (XTI) were submitted as base documents. A poll of users and developers indicated that a new transport layer interface to address the shortcomings of each of the pre-existing transport layer interfaces was not desired. There were strong proponents of both Sockets and XTI, making an either/or choice unfeasible.

P1003.1g currently consists of a low-level interface called the Detailed Network Interface (DNI). This low-level interface specifies both XTI and Sockets. The approach of a dual DNI interface was taken in an attempt to preserve the current investment in existing practice and to promote application portability. Both XTI and Sockets have been in use for many years and there are a large number of XTI and Socket programs.

The P1003.1g specification is proceeding through the IEEE ballot process. The first ballot closed October 1993. An approved standard is expected in 1996. The current draft is Draft 6.0 dated January 1995.

A P1003.1g conformant system will be required to support both XTI and Sockets. This requirement is important to ensure support for the large installed base of existing XTI and Sockets applications. Although XTI and Sockets provide similar functionality, the two interfaces are not identical. Sections 4 and 5 describe DNI/XTI and DNI/Sockets in more detail. There are many operating systems which currently provide both XTI and Sockets. Vendors which support both XTI and Sockets include Sun Microsystems, IBM, DEC, OSF, HP, and X/Open.

It is likely that a high-level interface called the Simple Network Interface (SNI) and a Naming Interface will be added in the future. The P1003.1g working group has postponed this work in order to complete the DNI work.

The DNI provides access to protocol-specific features of the underlying network by allowing the use of optional features unique to specific protocol families. Both connectionless

and connection-oriented transport services are supported. The current P1003.1g standard defines interfaces which provide access to the Internet family of protocols, the ISO/OSI family of protocols, and local interprocess communication (IPC). Since P1003.1g is protocol independent, P1003.1g may be used to access additional transport providers if the mappings to those protocols are defined.

Use of a P1003.1g API does not in itself guarantee interoperability between communicating processes. Interoperability is provided by matching communication stacks implemented beneath the 1003.1g API's, as well as the application protocol used above the 1003.1g API's.

P1003.1g specifies language independent interface (LIS) mappings to local interprocess communication (IPC), ISO Transport Protocols, and Internet Transport Protocols. For both DNI interfaces, P1003.1g specifies C binding interface mappings to ISO Transport Protocols and Internet Transport Protocols. For DNI/Sockets, there is also an interface mapping for local interprocess communication (IPC). Other language bindings may be defined in the future.

3 Common Programming Interface for Communications

In the 1980's, SNA provided a set of communications protocols for hierarchical communications between specific devices. The device types, known as Logical Unit (LU) types, are software that handle data communications. To support the distributed processing required by client/server applications, new LU types were developed. LU 6.2 was designed in 1984 as a generic protocol to support distributed peer-to-peer processing between different types of computers. The term APPC, which stands for Advanced Program-to-Program Communication, is commonly used instead of the term LU 6.2.

APPC is a communications protocol that enables programs on different computers to communicate. APPC lies between application programs and the network. The APPC architecture did not specify a common API for implementing functions. As a result, each operating system supporting APPC developed its own set of verbs. Programmers developing client/server applications for different operating systems were required to learn how to use two or more different sets of verbs. Common Programming Interface for Communications (CPI-C) was announced in 1987 as an easier to use, higher level, more tightly controlled interface to APPC. CPI-C, which sits on top of LU6.2, provides a consistent API for program-to-program communications across different platforms.

Both IBM and X/Open have published version of the CPI-C specification. Section 3.1 provides a listing of CPI-C 1.0, CPI-C 1.1, X/Open extensions to CPI-C 1.1, and CPI-C 1.2 functionality. The functionality added to each version of CPI-C is listed, but will not be described in this report. Consult one of the CPI-C documents listed in section 7 for more information.

IBM initiated the CPI-C Implementors' Workshop (CIW) to help guide and define future enhancements to the CPI-C specification. All users and implementors of CPI-C are encouraged to participate in the workshop. CPI-C 2.0 was developed in conjunction with the CIW. As of CPI-C 2.0, IBM and X/Open CPI-C specifications will be aligned. CPI-C 2.0 includes mappings for OSI Transaction Processing (TP) as well as LU 6.2. Similarities between SNA LU 6.2 and OSI's Transaction Processing (TP) services are well documented [17]. The OSI TP work is contained in the ISO/IEC 10026-series of standards for distributed transaction processing [6].

Section [4] describes CPI-C in more detail and highlights CPI-C 2.0 enhancements.

Unlike P1003.1g/DNI, CPI-C is not a transport layer interface. In the OSI Reference Model, CPI-C resides above the transport layer. In addition to providing the basic transport of data, CPI-C also provides numerous advanced functions. As a result, CPI-C is much more complex than the P1003.1g transport layer APIs. CPI-C functionality which provides more than the basic reliable end-to-end data transfer includes:

- Synchronization and control
- Conversation security

- Distributed directory service
- Distributed security service
- Resource recovery

Although P1003.1g/DNI and CPI-C provide many similar services, CPI-C does not provide access to protocol specific features. CPI-C by itself does not provide transport independence. CPI-C is commonly used for distributed databases and is designed to support a large number of clients in a single program.

CPI-C is language neutral. CPI-C provides one standard set of verbs, known as CPI-C calls, for all platforms that support CPI-C. This set of verbs is similar to a language independent specification. The calls can be made from most languages without any changes. Languages that can call on CPI-C include C, Cobol, Fortran, PL/1, Pascal, the CSP application generator, the REXX procedures language, and RPG.

Applications can obtain the services of CPI-C as implemented in AIX, CICS/ESA, DOS, IMS/ESA, MVS/ESA, OS/2, OS/400, and VM/ESA. Although CPI-C is available for most IBM systems, only a few other vendors have implemented CPI-C on their platforms.

3.1 Previous Versions of CPI-C

3.1.1 CPI-C 1.0

CPI-C Version 1.0 was originally published in 1988. This version contained dialogues and provided access to the conversation specific functionality of IBM's LU 6.2. CPI-C 1.0 provided a standard base for conversational communications including the following functionality:

- Ability to start and end conversations
- Support for program synchronization through confirmation flows
- Error processing
- Ability to optimize conversation flow

3.1.2 CPI-C 1.1

CPI-C 1.1 [8] was published in 1990 and added the following four areas to CPI-C 1.0:

- Support for resource recovery
- Automatic parameter conversion
- Support for communication with non CPI-C programs
- Local/remote transparency

3.1.3 X/Open CPI-C

CPI-C was first adopted by X/Open in 1990. The X/Open XPG4 version of CPI-C [3] is based upon IBM's CPI-C 1.1 and is know as X/Open Extensions to CPI-C. X/Open CPI-C was included within X/Open's Common Applications Environment (CAE).

CPI-C was primarily included within X/Open's CAE to define a standard basis for mainframe interworking. X/Open CPI-C was intended to provide an API that allows X/Open-compliant systems to communicate with systems that have implemented the IBM LU6.2 protocol [5].

Support for resource recovery was excluded from X/Open's XPG4 CPI-C definition. X/Open's version of CPI-C included several new functions not found in CPI-C 1.1:

- Support for non-blocking calls
- Ability to accept multiple conversations
- ASCII and EBCDIC conversions
- Support for security parameters

X/Open's Peer-to-Peer SnapShot was published as a requirements document for the CPI-C Implementor's Workshop. This document provides a detailed mapping of the Peer-to-Peer interface to the OSI TP Services.

3.1.4 CPI-C 1.2

CPI-C 1.2 consolidates CPI-C 1.1 and the X/Open extensions added to CPI-C 1.1. CPI-C 1.2 includes the following:

- Support for non-blocking calls
- Support for data conversion
- Support for specification of security parameters
- Ability to accept multiple conversations

As of CPI-C 1.2, IBM agreed to remove all licensing requirements for CPI-C specifications.

3.2 Current Version of CPI-C

The current version of CPI-C is Version 2.0. As of CPI-C 2.0, IBM and X/Open versions of CPI-C will be aligned. CPI-C 2.0 provides the following enhancements to CPI-C 1.2 [4]:

- 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

4 DNI/XTI

In the mid-1980s, AT&T introduced the Transport Layer Interface (TLI) with Release 3 of the System V UNIX operation system [7]. XTI is an adaptation of TLI and was originally defined in the X/Open Portability Guide, Issue 3 (XPG3). XPG3 was published in late 1988. XTI, which is very similar to TLI, removes the system dependencies from TLI. The current DNI/XTI work is based on the XPG4 January 1992 version of the X/Open CAE Specification: X/Open Transport Interface. It is the goal of the P1003.1g Working Group to keep DNI/XTI and X/Open XTI aligned.

XTI defines a protocol independent interface to the transport layer. XTI was designed primarily for use with the ISO Transport Service Definition, but has also been adapted for use with Internet protocols such as TCP and UDP. Unlike Sockets, XTI does not provide direct local interprocess communication or access to layers beneath the transport layer.

There is much support for XTI in the Unix community. Support for XTI includes Sun, Sys V, Unix International, and the Open Software Foundation, whose members include Apollo, DEC, HP, and IBM.

4.1 Overview of XTI Communication

Before data transfer can take place, a series of XTI function calls are needed to establish the communications endpoint. Figure 3 shows the sequence of XTI function calls for a client and server communicating in connectionless mode.

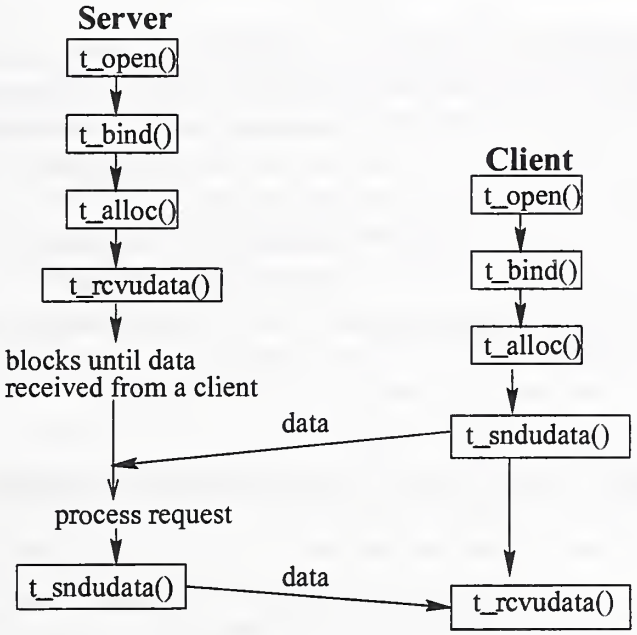

Figure 3: XTI Function Calls Using Connectionless Mode

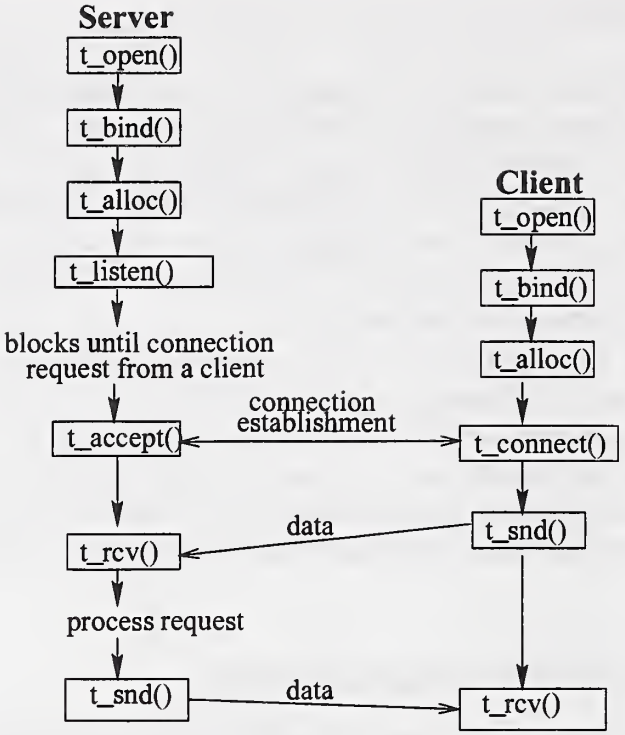

Figure 4: XTI Function Calls Using Connection-oriented Mode

Using connection-oriented communications, before data transfer can take place, a series of XTI function calls is needed to establish the connection between the two endpoints. Figure 4 shows the sequence of XTI function calls for a client and server communicating in connection-oriented mode. Sections 4.6 and 4.7 provide an overview of connection establishment and connection release.

Sections 4.2 and 4.3 provide a high level summary of the function calls shown in figures 3 and 4. These sections do not contain sufficient detail needed to write XTI programs. Section 7 provides a list of references that provide XTI programming examples. Note that TLI functions are very similar to DNI/XTI functions, but may deviate slightly. Consult the P1003.1g specification for detailed information on XTI functions.

4.2 Basic XTI Functions For Connectionless Mode Communication

A connectionless mode communications service has two phases of communication: initialization/de-initialization and data transfer. The following functions are used for basic connectionless mode communications.

1. Initialization/de-initialization phase

- t_open()

  The t_open() function creates a communications endpoint and returns protocol-specific information associated with that endpoint. This function also returns a file descriptor that serves as the local identifier of the endpoint.

- t_bind()

  The t_bind() function associates an address with a communications endpoint. Applications can either explicitly specify a end-point's address or permit the system to assign one.

- t_alloc()

  The t_alloc() function dynamically allocates memory for various communications function argument structures.

- t_close()

  The t_close() function informs the communications provider that the user is finished with the communications endpoint. Any local resources associated with that endpoint are freed.

2. Data transfer

   - t_sndudata()

     The t_sndudata() function enables communications users to send a self-contained data unit to the user at the specified protocol address.

   - t_rcvudata()

     The t_rcvudata() function enables communications users to receive data units from other users.

   - t_rcvuderr()

     t_rcvuderr() function enables communications users to retrieve error information associated with a previously sent data unit.

4.3 Basic XTI Functions For Connection-oriented Mode Communication

A connection-oriented communications service has four phases of communication: initialization/de-initialization, connection establishment, data transfer, and connection release. The following functions are used for basic connection-oriented mode communications. Each endpoint has certain characteristics associated with the communication. Section 4.8 lists communications options.

1. Initialization/de-initialization

- t_open()
  The t_open() function creates a communications endpoint and returns protocol-specific information associated with that endpoint. This function also returns a file descriptor that serves as the local identifier of the endpoint.
- t_bind()
  The t_bind() function associates an address with a communications endpoint.
- t_alloc()
  The t_alloc() function dynamically allocates memory for various communications function argument structures.
- t_close()
  The t_close() function informs the communications provider that the user is finished with the communications endpoint. Any local resources associated with that endpoint are freed.

2. Connection establishment

   - t_connect()
     The t_connect () function requests a connection to the communications user at a specified destination and waits for the remote user's response.
   - t_listen()
     The t_listen () function enables the passive communications user to receive connect indications from other communications users.
   - t_accept()
     The t_accept () function is issued by the passive user to accept a particular connect request after an indication has been received.

3. Data transfer

   - t_snd()
     The t_snd() function enables communications users to send either normal or expedited data over a connection.
   - t_rcv()
     The t_rcv() function enables communications users to receive either normal or expedited data over a connection.

4. Connection release

   - t_snddis()
     The t_snddis() function can be issued by either communications user to initiate the abortive release of a connection.

- t_rcvdis()
  The t_rcvdis() function identifies the reason for the abortive release of a connection, where the connection is released by the communication provider or another communications user.
- t_sndrel()
  The t_sndrel() function can be called by either communications user to initiate an orderly release. The connection remains intact until both users call t_sendrel() and t_rcvrel().
- t_rcvrel()
  The t_rcvrel() function is called when a user is notified of an orderly release request, as a means of informing the communications provider that the user is aware of the remote user's actions.

4.4 Modes of Service

DNI/XTI supports both connectionless and connection-oriented communication. One of the parameters to *t_open()* specifies the type of service provided by the transport provider. Three types of service that can be offered by a transport provider are: connection-oriented service, without orderly release; connection-oriented service, with orderly release; and connectionless service. Section 4.7 explains the concept of orderly release. DNI/XTI provides full-duplex data transfer.

4.5 Execution Modes

XTI calls are made in one of two execution modes: blocking, or non-blocking. The desired mode is specified through the O_NONBLOCK flag. This flag may be set with *t_open()* when the communications provider is initially opened. The O_NONBLOCK flag may also be set using the *fcntl()* function. When the O_NONBLOCK flag is clear, the function blocks until the processing associated with the function either completes or is interrupted by an event. When the O_NONBLOCK flag is set, the function does not block and initiates the processing associated with the function.

4.6 Overview of Connection Establishment

The function *t_open()* is called as the first step in the initialization of a communications endpoint. This function returns a file descriptor that is used by all other XTI functions. Information regarding the transport provider is returned to the caller.

After the communications endpoint has been created, the *t_bind()* function assigns an address to the endpoint. Consult the P1003.1g specification for detailed information on addressing. The *t_alloc()* function is used to dynamically allocate memory for communications function argument structures. The *t_open()*, *t_bind()*, and *t_alloc()* functions are called for both connectionless and connection-oriented communication.

For connection-oriented mode, the connection establishment phase enables two communications users to establish a connection between them. One process is considered active and initiates the conversation. The other process is considered passive and waits for a connection request. The process initiating a connection issues a *t_connect()* call. The process accepting a connection issues a *t_listen()* and then a *t_accept()* call. After the connection has been established, user data can be transferred.

4.7 Overview of Connection Release

Section 4.3 lists functions that support connection release for connection-oriented mode. The functions *t_snddis()* and *t_rcvdis()* are used for abortive release and the functions *t_sndrel()* and *t_rcvrel()* are used for orderly release.

When an abortive release is issued, there is no guarantee of delivery of user data. A communications user may initiate an abortive release either in the connection establishment phase or the data transfer phase. When issued in the connection establishment phase, the abortive release rejects a connection request. The communications provider may also initiate an abortive release, in which case both users are informed that the connection no longer exists.

The procedure for orderly release prevents the loss of data that may occur during an abortive release. If supported by the communications provider, orderly release may be invoked from the data transfer phase to enable two users to gracefully release a connection.

The orderly release is an optional feature of TCP. The ISO Connection-oriented Transport Service Definition only supports abortive release.

4.8 Use of Options

Some options are general XTI options while others are specific for each communications provider. All options have default values, however the values can be negotiated by an application. Many of the XTI functions contain an argument which is used to convey options between an application and the communications provider. The *t_optmgmt()* function enables an application to retrieve, verify or negotiate protocol options with the communications provider. There are six XTI level options. Consult the P1003.1g specification for descriptions of XTI options and information regarding their use.

5 DNI/Sockets

A *socket* is a general-purpose interprocess communication (IPC) mechanism useful for both network and stand-alone applications. Current DNI/Socket work is based on the 4.4BSD Socket interface from the University of California, Berkeley. The sockets mechanism was initially introduced in BSD 4.2 in Unix in 1981. The sockets model generalizes standard I/O functions. Sockets have been widely implemented in several environments. In BSD Unix, sockets are built into the kernel and are accessible by way of system calls. Non-BSD Unix

systems, as well as other operating systems such as MS-DOS, MacOS, DOS/Windows, and OS/2, provide sockets in the form of libraries.

Although sockets are used widely on a number of different systems, sockets interfaces are not necessarily consistent across all systems. The P1003.1g DNI/Sockets specification is important because it promotes alignment of implementations with variations to the Sockets interface.

DNI/Sockets provides interfaces to Internet and ISO Transport protocols. This API provides transport layer access as well as access to lower layers.

5.1 Overview of Sockets Communication

Before data transfer can take place, a series of Sockets function calls are needed to establish the socket. Figure 5 shows the sequence of Sockets function calls for a client and server communicating in connectionless mode. Section 5.2 provides a brief description for each function call shown in figure 5.

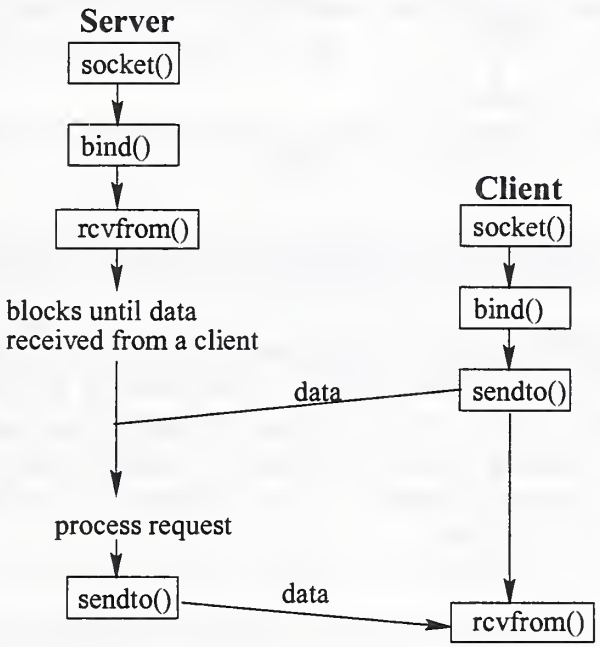

Figure 5: Sockets Function Calls Using Connectionless Mode

Using connection-oriented communications, before data transfer can take place, a series of Sockets function calls is needed to establish the connection between the sockets. Figure 6 shows the sequence of Sockets function calls for a client and server communicating in connection-oriented mode. Section 5.3 provides a brief description for each function call shown in figure 6. Sections 5.6 and 5.7 provide an overview of connection establishment and connection release.

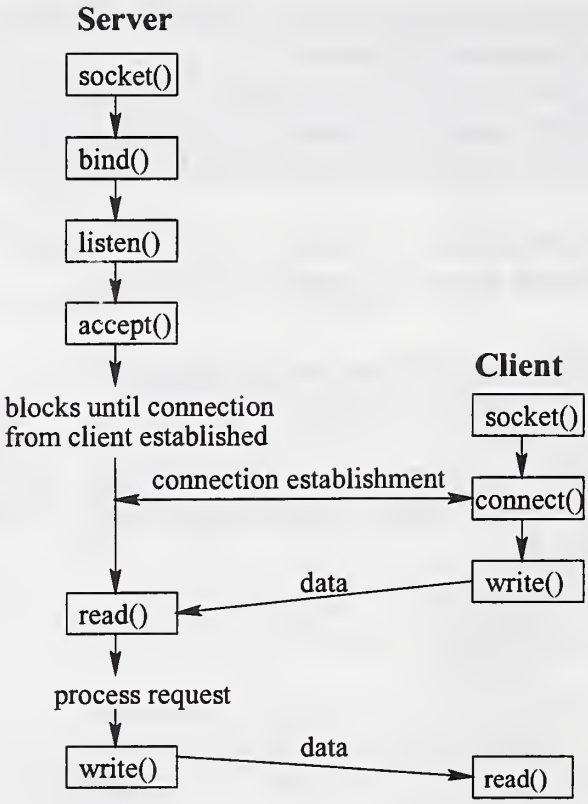

Figure 6: Sockets Function Calls Using Connection-oriented Mode

Sections 5.2 and 5.3 provide a high level summary of the function calls shown in figures 5 and 6. These sections do not contain sufficient detail needed to write Sockets programs, however, section 7 provides a list of references that provide Sockets programming examples. Note that although the P1003.1g Sockets interfaces is very similar to existing Sockets interfaces, some of the functions may slightly deviate. Consult the P1003.1g specification for detailed information on XTI functions.

5.2 Basic Sockets Functions for Connectionless Mode Communication

Sockets communicating in connectionless mode are know as datagram sockets. The datagram socket type supports connectionless data transfer which is not necessarily acknowledged or reliable. The following list contains functions that are used for datagram sockets. There is no requirement for connection establishment. Each message includes the destination address.

1. Socket creation

   - socket()

     The *socket()* function is used to create an endpoint for communication.

- bind()
  The *bind()* function may be called before the first data transmission to set a particular local address.

2. Data Transfer

   - sendto(), sendmsg()
     The *sendto()* and *sendmsg()* function may be called to send a message from a socket. These functions may be used regardless of whether a peer address has been prespecified.
   - recvfrom(), recvmsg()
     The *recvfrom()* and *recvmsg()* function may be called to send a message from a socket. These functions may be used regardless of whether a peer address has been prespecified.

If the *connect()* function is used to specify a peer address, the *send()* and *recv()* functions may also be used during the data transfer phase.

5.3 Basic Sockets Functions for Connection-Oriented Mode Communication

This section provides a high level summary of functions employing connection-oriented communication. A connection-oriented communications service has four phases of communication: initialization/de-initialization, connection establishment, data transfer, and connection release. The following functions are used for basic connection-oriented mode communications.

1. Initialization

   - socket()
     The *socket()* function is used to create a communications endpoint.
   - bind()
     The *bind()* function is used to bind an address to a socket.

2. Connection Establishment

   - connect()
     The *connect()* function requests a connection to the communications user at a specified destination and waits for the remote user's response.
   - listen()
     The *listen()* function enables the passive communications user to receive connect indications from other communications users.

- accept()
  The *accept()* function is issued by the passive user to accept a particular connect request after an indication has been received.

3. Data transfer

   - recv()
     The *recv()* function enables communications users to receive data over a connection.
   - send()
     The *send()* function enables communications users to send data over a connection.

4. Discarding Sockets

   - close()
     The *close()* function informs the communications provider that the user is finished with the communications endpoint. Any local resources associated with that endpoint are freed.
   - shutdown()
     The *shutdown()* function causes all or part of a full-duplex connection to be shut down. The connection may be shutdown so that either further receives will be disallowed, further sends will be disallowed, or further sends and receives will be disallowed.

5.4 Modes of Service

DNI/Sockets supports connectionless mode and connection-oriented mode. Local interprocess communication (IPC) is also supported. Local IPC involves the transfer of data between processes within the same system. Parameters to the *socket()* function calls specify the type of service desired. Section 5.6 briefly describes how types of sockets are selected.

5.5 Execution Modes

Like DNI/XTI, DNI/Sockets calls are made in either blocking or non-blocking mode. The desired mode is specified through the O_NONBLOCK flag. This flag may be set when the communications provider is initially opened. The O_NONBLOCK flag may also be set using the *fcntl()* function.

5.6 Overview of Connection Establishment

The function *socket()* is called as the first step in the initialization of a socket. This function returns a file descriptor that is used by all other Sockets functions. A socket is created with

a specific *type*, which defines the communication semantics and which allows the selection of an appropriate communications protocol. Four types of sockets are:

- SOCK_DGRAM (datagram socket)

  The SOCK_DGRAM socket type supports connectionless data transfer. Datagrams may be sent to a peer named in each output operation, and incoming datagrams may be received from multiple sources.

- SOCK_STREAM (stream socket)

  The SOCK_STREAM socket type provides reliable, sequenced, full-duplex octet streams between the socket and a peer to which the socket is connected. A socket of type SOCK_STREAM must be in a connected state before any data may be sent or received.

- SOCK_SEQPACKET (sequenced packet socket)

  The SOCK_SEQPACKET socket type is similar to the SOCK_STREAM type, and is also connection-oriented. The only difference between these types is that record boundaries are maintained using the SOCK_SEQPACKET type.

- SOCK_RAW (raw socket)

  The SOCK_RAW socket type is similar to the SOCK_DGRAM type. It differs in that it is normally used with communication providers that underly those used for the other socket types. For example, a SOCK_DGRAM socket can be used to access the network layer instead of the transport layer.

After the socket has been created, the *bind()* function assigns an address to the endpoint. Consult the P1003.1g specification for detailed information on addressing. The *socket()* and *bind()* functions are called for both connectionless and connection-oriented communication.

For connection-oriented mode, the connection establishment phase enables two sockets to establish a connection between them. One process is considered active and initiates the conversation. The other process is considered passive and waits for a connection request. The process initiating a connection issues a *connect()* call. The process accepting a connection issues a *listen()* and then an *accept()* call. After the connection has been established, user data can be transferred.

5.7 Overview of Connection Release

Section 6 lists sockets functions used for discarding sockets. Section 4.7 describes abortive and orderly release.

In the socket connection establishment phase, closing a socket with the *close()* function causes any queued but unaccepted connections to be discarded. In the data transfer phase, if the socket being closed is associated with a protocol that promises reliable delivery of data (i.e. orderly release), the socket must attempt to transmit the data.

A socket can be marked explicitly to force the application process to attempt to transmit pending data. If the socket is connection-oriented, the socket is connected, the SO_LINGER option is on with a non-zero linger time, and the O_NONBLOCK option is not set for the socket, the *close()* function will block until the disconnect completes or until the linger time expires. If the linger time expires before the disconnect is complete, any pending data is discarded.

The *shutdown()* function causes all or part of a full-duplex connection to be shut down. The connection may be shutdown so that either further receives will be disallowed, further sends will be disallowed, or further sends and receives will be disallowed.

5.8 Use of Options

There are several socket options which either provide useful information or specialize the behavior of a socket. Socket options are manipulated by two functions, *getsockopt()* and *setsockopt()*. These functions allow an application program to customize the behavior and characteristics of a socket to provide the desired effect. There are 15 socket level options. Consult the P1003.1g specification for descriptions of XTI options and information regarding their use.

6 CPI-C 2.0

Both IBM and X/Open have published version of the CPI-C specification. Section 3.1 provides a listing of CPI-C 1.0, CPI-C 1.1, X/Open extensions to CPI-C 1.1, and CPI-C 1.2 functionality. The functionality added to each version of CPI-C is listed, but will not be described in this report. Consult one of the CPI-C documents listed in section 7 for more information.

CPI-C 2.0 was developed at the CPI-C Implementor's Workshop and was published in 1994. As of CPI-C 2.0, IBM and X/Open versions of CPI-C will be aligned. This section provides an overview of CPI-C 2.0, similar to the overviews provided for P1003.1g DNI/XTI and P1003.1g DNI/Sockets.

6.1 Overview of CPI-C Communication

The part of a communications application that initiates or responds to APPC communications is called a *transaction program.* Transaction programs use verbs to communicate with each other. The verbs are used to start, stop, and control conversations. APPC verbs make up the API for APPC. These verbs represent the interface between the transaction program and the APPC software.

CPI-C is connection oriented. Connectionless mode communications is not supported. Before data transfer can take place, a series of CPI-C function calls is needed to establish the connection between the two endpoints. Figure 7 shows the sequence of CPI-C function calls for a client and server communicating in connection-oriented mode. Section 6.2 provides a brief description for each function call shown in figure 7. Sections 6.6 and 6.7 provide an overview of connection establishment and connection release. Function names correspond to CPI-C verbs. As displayed in the CPI-C code shown in appendix C, function names such as *Initialize_Conversation* are not used in CPI-C programs. Instead, CPI-C programs use six-character short names. For example, the function name *cminit()* is used instead of the function name *Initialize_Conversation()*. Short function names are used because CPI-C calls are designed to be the same across multiple languages, platforms, and programming environments, some of which have name-length restrictions.

6.2 Basic Calls for CPI-C Communication

CPI-C programs communicate by making program calls. CPI-C is connection-oriented and CPI-C programs exchange data using a conversation. Program calls are used to establish the characteristics of the conversation and to exchange data and control information. CPI-C maintains a set of characteristics for each conversation used by a program. Characteristics are established for each program on a per-conversation basis.

CPI-C calls can be categorized into two groups of functions, starter-set calls and advanced-function calls. The starter-set group of calls consists of six functions. These calls are used for simple communication of data between two programs. Programs using the starter-set of

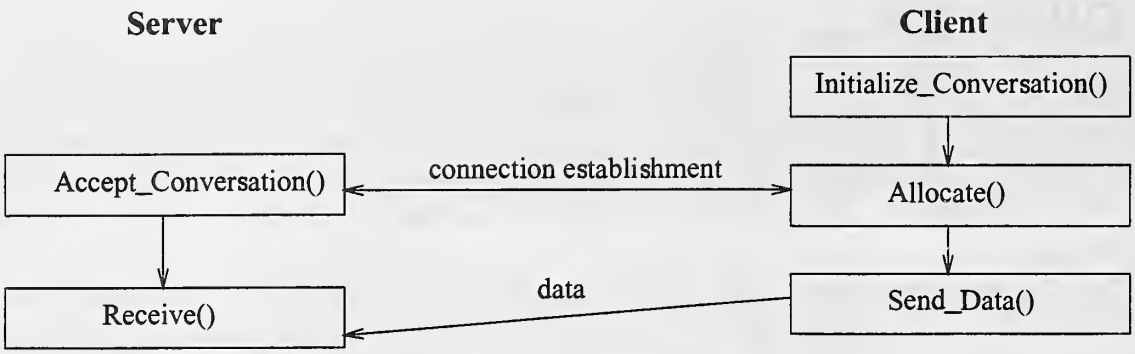

Figure 7: CPI-C Function Calls Using Connection-oriented Mode

calls do not require values for CPI-C conversation characteristics that vary from the initial values. The advanced-function group of calls consists of over 40 functions. These calls are used to do more specialized processing. Characteristic values other than the default values may be specified. The advanced-function calls can be used to provide careful synchronization and monitoring of data.

The following is a list of starter-set calls. A brief description of each call is provided. The actual syntax used to program the calls depends on the programming language used. Consult the CPI-C specification for detailed information on CPI-C functions.

- Initialize_Conversation()

  A program issues the *Initialize_Conversation* call to prepare to start the conversation.

- Allocate()

  The program initializing a conversation uses the *Allocate* call to start the conversation. Depending on the *conversation_type* characteristic, either a basic or mapped conversation is established.

- Accept_Conversation()

  The partner program uses the *Accept_Conversation* call to initialize values for the conversation characteristics on its end of the conversation. When a *Accept_Conversation* call is issued, the partner program receives a unique *conversation_ID*.

- Send_Data()

  The *Send_Data* call is used to send data between programs. When this call is issued during a mapped conversation, one data record is sent to the partner program. When this call is issued during a basic conversation, data sent to the partner program consists of logical records.

- Receive()

  The Receive call is used to receive information from a given conversation. The information received can be data, a data record, conversation status, or a request for confirmation or for SAA resource recovery services.

- Deallocate()

  The *Deallocate* call is used to end a conversation.

CPI-C supports two modes for sending and receiving data on a conversation: half-duplex and full-duplex. For half-duplex conversation, only one of the programs has the right to send data at any time. Send control must be transferred to the other program before that program can send data. For full-duplex conversation, both programs can send and receive data at the same time. The sending and receiving mode for a conversation is determined at the time the conversation is established.

6.3 Conversation Characteristics

CPI Communications maintains a set of characteristics for each conversation used by a program. These 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. Initial values for conversation characteristics are set by the *Initialize_Conversation*, *Accept_Conversation*, *Initialize_For_Incoming*, and *Accept_Incoming* calls. There are over 40 different conversation characteristics. There are even more calls to set and extract conversation characteristic values. Examples are *Extract_Conversation_Type* and *Set_Conversation_Type*. Consult the CPI-C specification for descriptions of conversation characteristics and information on their use.

6.4 Modes of Service

Both SNA LU6.2 and OSI TP are connection-oriented. CPI-C only supports connection-oriented communication. Two types of conversations are supported:

- Mapped conversations allow programs to exchanged data records in data formats agreed upon by the application programmers.

- Basic conversations allow programs to exchange data in a standardized format.

CPI-C 1.2 only supported half-duplex conversation and did not support full-duplex conversation. CPI-C 2.0 supports both half-duplex and full-duplex conversation. In CPI-C 2.0, a new conversation characteristic, *send_receive_mode* is defined with two values, CM_FULL_DUPLEX and CM_HALF_DUPLEX. A programmer specifies either full-duplex or half-duplex conversation by using the *Set_Send_Receive_mode* call.

6.5 Modes of Execution

CPI-C calls can be made in either blocking or non-blocking mode. CPI-C 2.0 extends non-blocking support beyond the conversation level, to the queue level. Queue-level non-blocking means: the processing mode is set on a queue basis, only one outstanding operation is allowed per conversation queue. Conversation queues are logical communication channels between the program and CPI-C. Conversation queues are introduced in CPI-C 2.0 to allow: the independent operations required for full-duplex conversations, more than one outstanding operation on a conversation. The concept of conversation queue applies to both full-duplex and half-duplex conversations.

6.6 Overview of Connection Establishment

The *Initialize_Conversation* call is used to establish a conversation. The program initiating a conversation uses the *sym_dest_name parameter* to designate a program partner and receives a unique conversation identifier, the *conversation_ID*. The *Initialize_Conversation* call initializes values for various conversation characteristics before the conversation is allocated.

6.7 Overview of Connection Release

The *Deallocate* call is used to end a conversation. When the conversation is deallocated, the *conversation_ID* is no longer associated with the conversation. The default use of the *Deallocate* call is for CPI-C to return immediately to the program without waiting for the partner to acknowledge that it's ready to end the conversation. One *Deallocate* call ends the conversation for both sides. The *Deallocate* call can be used in conjunction with the *Confirm* call to perform a confirmation at the end of the conversation.

7 Summary

The P1003.1g APIs and CPI-C are capable of process to process communications. This section characterizes the types of applications that make use of each of these APIs. Some of the advantages and disadvantages of using each of the APIs are presented. Figure 8 compares general features of P1003.1g and CPI-C.

Because sockets have been available longer than XTI, most existing network applications use sockets. Furthermore, the majority of these applications use TCP or UDP rather than ISO transports since TCP/IP was available about 10 years before ISO transport. As a result of the large installed base of network applications implemented with sockets, there is a large community of programmers with expertise in socket programming.

The majority of network applications that are implemented with XTI use TCP or UDP. Applications requiring ISO transport tend to use XTI rather than sockets.

There are strong proponents for both XTI and Sockets and there are vast numbers of existing XTI and Sockets programs. Many application developers have a preference for one of the two interfaces. Although many systems support both Sockets and XTI, some systems do not fully support both interfaces. Programmers are likely to use the interface supported by their system. P1003.1g compliant systems will be required to fully support both Sockets and XTI.

	P1003.1g	**CPI-C**
Type of API	transport layer	above transport layer
Type of data transfer	full-duplex	half or full-duplex
Type of Service	connection-oriented or connectionless	connection-oriented
Protocol Independent?	Yes	not by itselft, but can be in conjunction with IBM's MPTN
Advanced functions?	No	Yes

Figure 8: Comparison of P1003.1g and CPI-C Features

IBM's APPC programs implementing SNA's LU6.2 were introduced in 1982. The user base for LU6.2 has been slowly growing since 1982. CPI-C was announced in 1987. CPI-C 2.0 sits on top of LU 6.2 and OSI TP. CPI-C is commonly used for distributed databases and transaction processing.

The P1003.1g APIs are transport layer APIs. They provide the basic service of reliable, end-to-end data transfer needed by applications and higher layer protocols. CPI-C resides above the transport layer. In addition to providing the basic transport of data, CPI-C also provides numerous advanced functions such as synchronization and control, conversation security, distributed directory service, distributed security service, and resource recovery. CPI-C is a much more complex interface the P1003.1g APIs. A fundamental difference between the P1003.1g APIs and CPI-C is that the P1003.1g APIs provide a protocol independent interface. The P1003.1g APIs provide access to protocol-specific features of the underlying network in a protocol independent manner.

An advantage of DNI/Sockets is that it offers defined transport, local IPC, and data link access. A disadvantage of DNI/Sockets is that because of the variations with socket support among vendors, many pre-existing socket applications are not directly portable across platforms. For example, many Sockets implementations have different socket-length fields (i.e, 8 bit or 16 bit).

An advantage of XTI is that users concerned about migrating their applications to ISO transport may use DNI/XTI since DNI/XTI has similar semantics for both TCP/IP and ISO transport specific features. A disadvantage of using DNI/XTI is that DNI/XTI is transport specific, and currently does not provide support for local IPC.

The P1003.1g DNI/XTI and DNI/Sockets interfaces can be used for connection-oriented and connectionless network programming. The DNI APIs are transport layer APIs and provide access to protocol specific features of the underlying network in a protocol independent manner. Although both DNI APIs support migration between Internet and OSI protocols, if an application wants to run easily over both Internet and OSI protocols, DNI/XTI is a better choice. Applications which need to gain access to the Data Link layer or do local IPC will use DNI/sockets. DNI/XTI does not provide such access because DNI/XTI is transport specific.

CPI-C is the choice for applications using distributed databases. CPI-C is an application layer API which does not provide access to protocol specific features of the underlying network. CPI-C does not provide connectionless mode communication, but does include more advanced functions such as security, synchronization, and error reporting. CPI-C by itself does not provide transport independence.

References

[1] IEEE P1003.12 Protocol Independent Interfaces for Process-to-Process Communication. Draft 4.1. February 1994.

[2] Huntley, Tim, "CPI-C Or Sockets: Choosing A Communications API", *IBM Internet Journal* September 1993, Volume 1, Number 9, p. 40.

[3] Common Programming Interface Communications, X/Open CAE Specification C210, ISBN 1-872630-35-9, March 1992.

[4] Common Programming Interface Communications Specification CPI-C 2.0, Second Edition, Document Number SC31-6180-00, June 8, 1994.

[5] "An Introduction to X/Open and XPG4", X/Open Briefing Papers, Set No. 3. Part 6: Mainframe Interworking, September 1993.

[6] ISO/IEC 10026, Information Technology - Open Systems Interconnection - Distributed Transaction Processing.

[7] AT&T, *UNIX System V Release 3.2 - Programmer's Reference Manual*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[8] CPI-C 1.1, SAA Common Programming Interface: Communications Reference, seventh edition, SC26-4399-06, 1992.

[9] Stallings, William, *Data and Computer Communications*, Macmillan Publishing Company, 1991.

[10] Comer, Douglas E. and Stevens, David L, *Internetworking with TCP/IP Volume III: AT&T TLI Version*, Prentice Hall, 1994.

[11] Stevens, W. Richard, *Unix Network Programming*, Prentice Hall, 1990.

[12] Piscitello, David M. and Chapin, A. Lyman, *Open Systems Networking: TCP/IP and OSI*, Addison-Wesley Publishing Company, 1993.

[13] Tanenbaum, Andrew S., *Computer Networks*, Prentice Hall, 1989.

[14] Leffler, Samuel J, et al, *The Design and Implementation of the 4.3 BSD Unix Operating System*, Addison-Wesley Publishing Company, 1990.

[15] "Getting Started with CPI-C Part I", *The APPC Connection*, Volume 3, Issue 4, July/August 1994.

[16] "Getting Started with CPI-C Part II, *The APPC Connection*, Volume 3, Issue 6, November/December 1994.

[17] Fetvedt, John, "Mapping Communications (1.2) onto OSI Distributed Transaction Processing Services", IBM, April 8, 1993.

[18] Fetvedt, John, "Proposed Enhancements to CPI-C for Support of OSI Distributed Transaction Processing", IBM, May 12, 1993.

[19] Walker, John and Schwaller, Peter, *CPI-C Programming in C: An Application Developer's Guide to APPC*, McGrawHill, 1994.

A XTI Programming Example

```
/*
 * Example of client using TLI and the TCP protocol.
 * Example program was taken from reference [10]
 */

#include "inet.h"

main(argc, argv)
int argc;
char *argv[];
{
int tfd;
char *t_alloc();
struct t_call *callptr;
struct sockaddr_in serv_addr;

pname = argv[0];

/*
 * Create a TCP transport endpoint and bind it.
 */

if ( (tfd = t_open(DEV_TCP, O_RDWR, 0)) < 0)
err_sys("client: can't t_open %s", DEV_TCP);

if (t_bind(tfd, (struct t_bind *) 0, (struct t_bind *) 0) < 0)
err_sys("client: t_bind error");

/*
 * Fill in the structure "serv_addr" with the address of the
 * server that we want to connect with.
 */

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port        = htons(SERV_TCP_PORT);
```

```
/*
 * Allocate a t_call structure, and initialize it.
 * Let t_alloc() initialize the addr structure of the t_call structure.
 */

if ( (callptr = (struct t_call *) t_alloc(tfd, T_CALL, T_ADDR)) == NULL)
err_sys("client: t_alloc error");
callptr->addr.maxlen = sizeof(serv_addr);
callptr->addr.len    = sizeof(serv_addr);
callptr->addr.buf    = (char *) &serv_addr;
callptr->opt.len     = 0; /* no options */
callptr->udata.len   = 0; /* no user data with connect */

/*
 * Connect to the server.
 */

if (t_connect(tfd, callptr, (struct t_call *) 0) < 0)
err_sys("client: can't t_connect to server");

doit(stdin, tfd);

close(tfd);
exit(0);
}

/*
 * Read the contents of the FILE *fp, write each line to the
 * transport endpoint (to the server process), then read a line back from
 * the transport endpoint and print it on the standard output.
 */

doit(fp, tfd)
register FILE *fp;
register int tfd;
{
int n, flags;
char sendline[MAXLINE], recvline[MAXLINE + 1];

while (fgets(sendline, MAXLINE, fp) != NULL) {
n = strlen(sendline);
if (t_snd(tfd, sendline, n, 0) != n)
```

```
err_sys("client: t_snd error");

/*
 * Now read a line from the transport endpoint and write it to
 * our standard output.
 */

n = t_rcv(tfd, recvline, MAXLINE, &flags);
if (n < 0)
err_dump("client: t_rcv error");
recvline[n] = 0; /* null terminate */
fputs(recvline, stdout);
}

if (ferror(fp))
err_sys("client: error reading file");
}

/*
 * Example of server using TLI and the TCP protocol.
 * Example taken from reference [10]
 */

#include "inet.h"

main(argc, argv)
int argc;
char *argv[];
{
int tfd, newtfd, clilen, childpid;
struct sockaddr_in cli_addr, serv_addr;
struct t_bind req;
struct t_call *callptr;

pname = argv[0];

/*
 * Create a TCP transport endpoint.
 */

if ( (tfd = t_open(DEV_TCP, O_RDWR, (struct t_info *) 0)) < 0)
err_dump("server: can't t_open %s", DEV_TCP);
```

```
/*
 * Bind our local address so that the client can send to us.
 */

bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sin_family      = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port        = htons(SERV_TCP_PORT);

req.addr.maxlen = sizeof(serv_addr);
req.addr.len    = sizeof(serv_addr);
req.addr.buf    = (char *) &serv_addr;
req.qlen        = 5;

/*
if (t_bind(tfd, &req, (struct t_bind *) 0) < 0)
err_dump("server: can't t_bind local address");
*/
if (t_bind(tfd, NULL, NULL) < 0)
err_dump("server: can't t_bind local address");

/*
 * Allocate a t_call structure for t_listen() and t_accept().
 */

if ( (callptr = (struct t_call *) t_alloc(tfd, T_CALL, T_ADDR)) == NULL)
err_dump("server: t_alloc error for T_CALL");

printf ("beginning to wait for incomming connections\n");
for ( ; ; ) {
/*
 * Wait for a connection from a client process.
 * This is an example of a concurrent server.
 */

if (t_listen(tfd, callptr) < 0)
err_dump("server: t_listen error");

if ( (newtfd = accept_call(tfd, callptr, DEV_TCP, 1)) < 0)
err_dump("server: accept_call error");

if ( (childpid = fork()) < 0)
```

```
err_dump("server: fork error");

else if (childpid == 0) { /* child process */
t_close(tfd); /* close original endpoint */
str_echo(newtfd); /* process the request */
exit(0);
}

close(newtfd); /* parent process */
}
}
```

B Sockets Programming Example

```
/*-------------------------------------------------------------------
 * Sockets "Hello, world" program
 * Client side
 *-------------------------------------------------------------------*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/errno.h>

#define ERROR(s)    {fprintf(stderr,"%d-",errno); perror(s); return(-1);}

char *msg = "Hello World";
int errno;

main(argc,argv)
int argc; char *argv[];
{
int consocket;                /* socket to connect into the socket */
struct sockaddr_in sa; /* mode, addr, and port data for the socket */
struct hostent *remote_host;          /* internet numbers, names */

    if (argc != 2)  {
        fprintf (stdout, "usage: %s hostname\n", argv[0]);
        exit (-1);
    }

    /* create an internet style connection oriented socket */
    if ((consocket = socket( AF_INET, SOCK_STREAM, 0 )) < 0 )
    ERROR ("socket");

    /* set addressing information and port to use */
    if ((remote_host = gethostbyname(argv[1])) == (struct hostent *)NULL )
    ERROR("gethostbyname");
    sa.sin_family = AF_INET;
    (void) bcopy((char *)remote_host->h_addr, (char *) &sa.sin_addr,
                 remote_host->h_length );
    sa.sin_port = htons(22222);
```

```
    /* connect to the other end-point */
    if (connect(consocket,(struct sockaddr *)&sa, sizeof sa) < 0 )
        ERROR ("connect");

    /* write Hello World message to other process */
        if ( write(consocket, msg, strlen(msg) ) < 0 )
    ERROR ("write");

    /* all the socket connections are closed automatically */
    exit(0);
}

#include <stdio.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include <sys/param.h>
#include <errno.h>

#define ERROR(s)       {fprintf(stderr,"%d-",errno); perror(s); return(-1);}

main (argc, argv)
int argc;
char *argv[];
{
struct sockaddr_in sa;
struct sockaddr_in caller;
int s, msgsock, length;
int retval;
char buf[BUFSIZ], acknowledgement[BUFSIZ];



if ((s = socket(AF_INET,SOCK_STREAM,0)) < 0)
ERROR ("socket");

/* name socket, register the socket */
sa.sin_family= AF_INET;
sa.sin_addr.s_addr = INADDR_ANY;       /* not choosy about who calls */
        sa.sin_port= 22222;                    /* this is our port number */
```

```
        if (bind(s,(struct sockaddr_in *)&sa,sizeof sa) < 0)
                ERROR ("bind");                 /* bind address to socket */

/* get assigned port number and print it */
length = sizeof(sa);
        if (getsockname(s, (struct sockaddr_in *)&sa, &length) < 0)
                ERROR("getsockname");



/* listen for connections on a socket */
listen (s, 5);

/* Use accept to wait for calls to the socket.  Accept returns
a new socket which is connected to the caller.  msgsock will be a
      temporary (non-resuable) socket different from s */
if ((msgsock = accept(s,(struct sockaddr *)&caller,&length)) < 0)
       ERROR ("accept");

   /* optional ack; demonstrates bidirectionality */
    gethostname(buf, sizeof buf);

    /* write into the msgsock; the "s" is _only_for_rendezvous_ */
    if ( write ( msgsock, acknowledgement, sizeof acknowledgement ) < 1 )
        perror(argv[0]);


    /* read lines until the stream closes */

    retval = 1;
    while(retval !=0 )  {
       bzero(buf, sizeof(buf));
        if( (retval = read(msgsock, buf, sizeof(buf))) < 0)
                perror("reading stream message");
if (retval == 0) printf("ending connection\n");
        else  printf("read-->%s\n", buf);
 }

    close(msgsock);
    close (s);
```

```
    exit (0);
}
```

C CPI-C Programming Example

```
/*-------------------------------------------------------------
 *  CPI-C "Hello, world" program
 *  Client side (file HELLO1.C)
 *
 *  This code was taken from the book CPI-C Programming in C:
 *  An Application Developer's Guide to APPC by John Q. Walker
 *  and Peter J. Schwaller, published by McGrawHill
 *-------------------------------------------------------------*/
#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*)"HELLO1S "

/* 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];
    CM_RETURN_CODE  cpic_return_code;
    unsigned char * data_buffer = SEND_THIS;
    CM_INT32        send_length =
                            (CM_INT32)strlen((char *)SEND_THIS);

    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 */

    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);
}

/*-------------------------------------------------------------
 *  CPI-C "Hello, world" program
 *  Server side (file HELLO1D.C)

 *  This code was taken from the book CPI-C Programming in C:
 *  An Application Developer's Guide to APPC by John Q. Walker
 *  and Peter J. Schwaller, published by McGrawHill

 *-------------------------------------------------------------*/
#include <cpic.h>               /* conversation API library    */
#include <stdio.h>              /* file I/O                    */
#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 the null */
    (void)printf("%s\nPress a key to end the program...\n",
                 data_buffer);

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

    return(EXIT_SUCCESS);
}
```