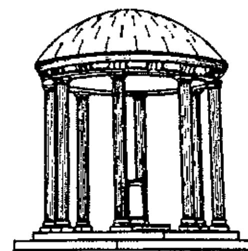
# A Common Network Interface for Interprocess Communication

*TR90-030*

*July, 1990*

*Debashish Chatterjee*

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



*UNC is an Equal Opportunity/Affirmative Action Institution.*

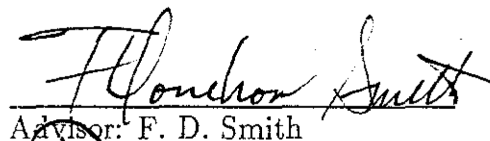
# A COMMON NETWORK INTERFACE
## FOR INTERPROCESS COMMUNICATION

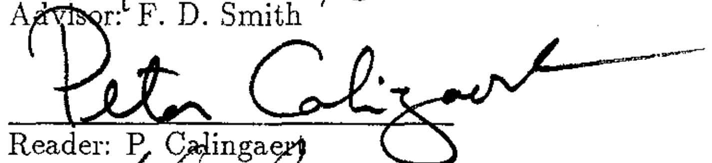
*by*
Debashish Chatterjee

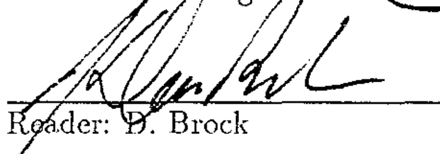
A thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill
July 1990

Approved by:


Advisor: F. D. Smith


Reader: P. Calingaert


Reader: D. Brock

©1990
Debashish Chatterjee
ALL RIGHTS RESERVED

**Debashish Chatterjee. A Common Network Interface for Interprocess Communication**
**(Under the direction of F. D. Smith)**

## ABSTRACT

This thesis describes a new programming interface to provide interprocess communication services independent of the network architectures and protocols available to support them. The common interface can be implemented using existing interfaces for a specific environment. The interface user is unaware of the specific protocol being used, even in a multi-protocol environment. This approach solves some problems of portability and interoperability. An implementation of this design on two network architectures has been completed. The performance and the limitations of this implementation are discussed. Future work in this area will also be discussed.

**Acknowledgement**

My special thanks to Don Smith for his help and encouragement through the course of this research;
to Peter Calingaert and Dean Brock for taking time to serve on the thesis committee and for their comments on the text;
and to Ashit Patel, Don Stone and Zhenxin Wang who helped in testing the design and implementation by building a messaging library using this interface.

# Contents

**List of Figures** vii

**1 Introduction** 1

1.1 The Problem . . . . . . . . . . 1

1.2 Related work . . . . . . . . . . 5

1.3 Common network interface approach . . . . . . . . . . 7

**2 Common Network Interface Design** 9

2.1 Existing models . . . . . . . . . . 9

2.2 Naming and binding . . . . . . . . . . 10

2.3 The common network interface features . . . . . . . . . . 12

**3 Common Network Interface Calls** 16

3.1 Creation, destruction, management . . . . . . . . . . 17

3.2 Data transfer and multiplexing . . . . . . . . . . 22

3.3 Utilities . . . . . . . . . . 26

3.4 Directory service . . . . . . . . . . 27

**4 Results and Conclusions** 28

4.1 Results . . . . . . . . . . 28

4.2 Conclusions and future work . . . . . . . . . . 30

**Bibliography** 33

**A Network Interface Implementation** 36

A.1 Some connection-oriented protocols . . . . . . . . . . . . . . . . 36

A.1.1 Transmission Control Protocol (TCP) . . . . . . . . . . 36

A.1.2 OSI Transport Layer (TP4) . . . . . . . . . . . . . . . . 37

A.1.3 APPC . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

A.2 Data objects . . . . . . . . . . . . . . . . . . . . . . . . . . 38

A.3 Code outline . . . . . . . . . . . . . . . . . . . . . . . . . . 44

**B Compiling and Linking the Interface Calls** 50

**C An Example Responder** 51

**D An Example Initiator** 58

**E Source Code Listing** 63

E.1 Include files and global declarations . . . . . . . . . . . . . . . 63

E.2 Creation, destruction, management . . . . . . . . . . . . . . . . 76

E.3 Data transfer and multiplexing . . . . . . . . . . . . . . . . . . 164

E.4 Utilities . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 220

# List of Figures

1.1 A heterogeneous environment . . . . . . . . . . . . . . . . . . . 3

4.1 Average timings . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.2 Best and worst timings . . . . . . . . . . . . . . . . . . . . . . 29

# Chapter 1

# Introduction

## 1.1 The Problem

Software designed for a networked computing environment typically consists of several processes. The processes do not necessarily execute on or use resources of a single machine. They do, however, need to communicate and synchronize among themselves to achieve the functions of the software. A software developer typically uses a set of services which provide reliable communication management and synchronization among independently executing processes. A programming interface provides primitives which give the programmer access to services.

The programming interfaces available depend upon the operating system, the network architecture, and the network protocols. A network architecture is a specific design for interconnecting computers. It defines the protocols to be used between the members of the network for meaningful information exchange, the data representation for the information exchange, and the addressing mechanism used to unambiguously reference a member. The operating system may support one or more network architectures and a network architecture, in turn, may support one or more protocols. Protocols supported are always specific to a network architecture.

Consider the Unix operating system. The Berkeley version of Unix sup-

ports a programming interface using *sockets*, while System V provides an interface called *Transport Library Interface* (TLI) [1] . Both support the Internet network architecture [14]. In addition, they may also support architectures like Open Systems Interconnection (OSI) [11, 12], Xerox XNS [7], Appletalk and others. Internet network architecture supports two protocols, Transmission Control Protocol (TCP) [15] and User Datagram Protocol (UDP) [13].

In a homogeneous environment, where the network architecture and the programming interface supported are uniform, processes use exactly this interface and protocol to achieve interprocess communication. The design is not complicated at all. However, consider the following heterogeneous scenario (See Fig. 1.1).

- Process $A_1$ running on machine $M_1$ uses interfaces $I_1$, $I_2$ and $I_3$.
- Process $A_2$ running on machine $M_2$ uses interfaces $I_1$ and $I_2$.
- Process $A_3$ running on machine $M_3$ uses interfaces $I_1$ and $I_3$.
- Process $A_4$ running on machine $M_4$ uses an interface $I_3$.

Let us assume interfaces $I_1$, $I_2$ and $I_3$ provide accesses to services provided by protocols $P_x$, $P_y$ and $P_z$ respectively. Mutual exchange of information between processes is possible *if and only if* they use a common protocol using the interface(s) available at their machine. For processes not having access to a common protocol, protocol conversion has to take place before communication can be established. Protocol conversion is the provision of augmented services to establish meaningful exchanges between two incompatible sets of rules, where each set by itself defines a format for meaningful exchange.

If processes $A_1$ and $A_2$ decide to interact, they may do so using either protocol $P_x$ or $P_y$. When $A_1$ communicates with $A_3$, $P_x$ or $P_z$ may be used. Process $A_3$ can communicate with either of $A_2$ or $A_4$ using only one specific protocol, which is again different for the two pairs. Processes $A_2$ and $A_4$ will

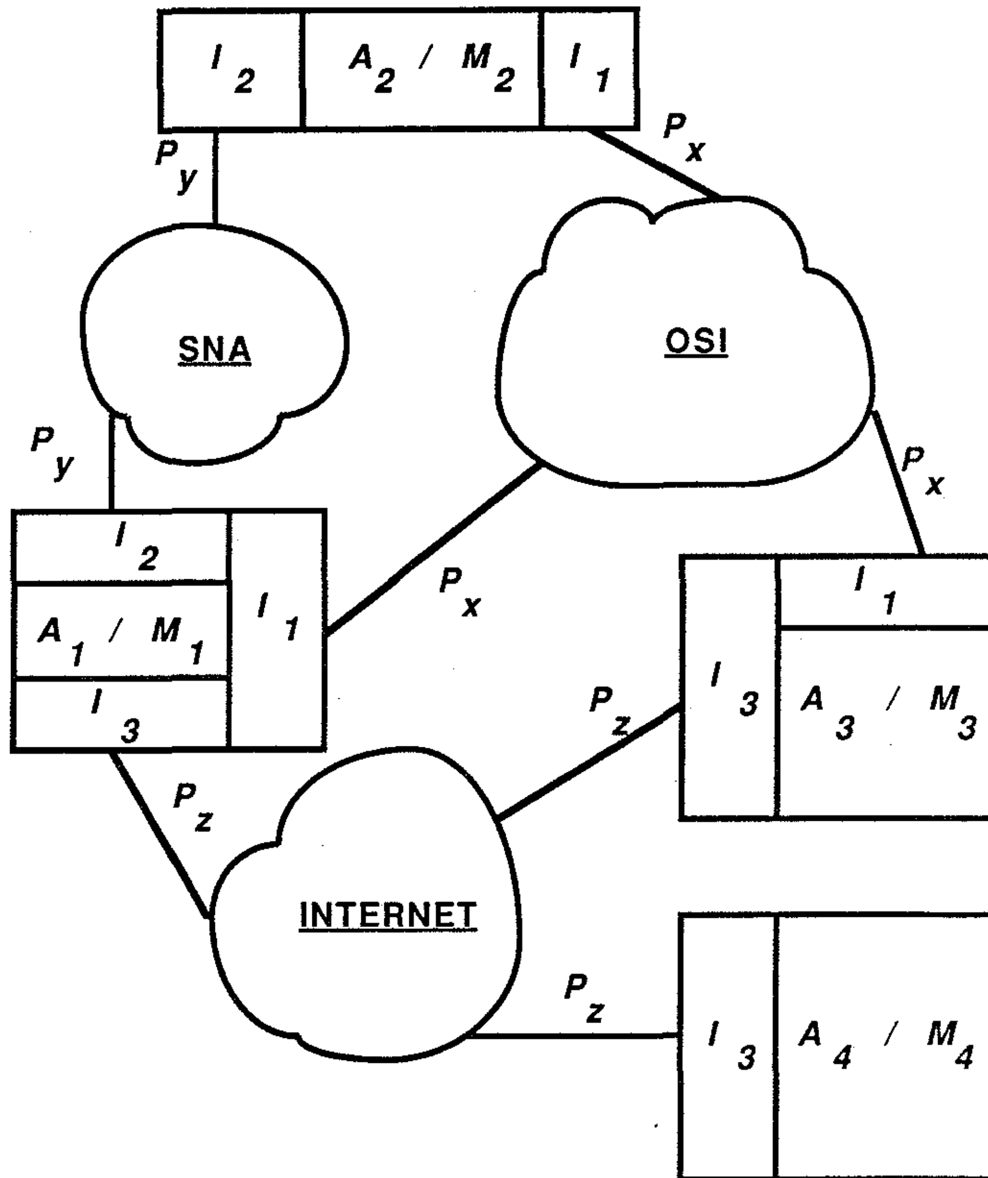



Figure 1.1: A heterogeneous environment

not be able to communicate unless protocol conversion takes place. Even if processes can communicate, the design of the programs will vary from one to another if the programming interfaces differ.

Consider the set of processes, shown in Figure 1.1, to make up an application. The scenario defines an environment where an individual process of the application explicitly uses a specific protocol and interface to talk to another process. The application designer has a complicated design at hand, depending upon the number of different interfaces available and the network architectures supported by these interfaces.

Very often the interface design is influenced by the design of a network architecture and the protocols supported by that architecture. This makes it unique and different from other network programming interfaces. The syntax of the programming interface defines the use of interface primitives; its semantics defines the services offered. Any substantial difference in syntax and the semantics of programming interfaces does not allow an application using a particular interface to use another without significant modification to the design of the application. If however, the syntactic flavor of the interfaces is done away with, many of these sets of primitives are quite similar in their semantics.

In this thesis, we give an interface definition providing a uniform set of external behaviors, irrespective of the actual protocol used to support the interface. This simplifies application design and development. It also solves some of the problems of portability and interoperability. Portability is affected by the available programming interfaces at the machines, while interoperability is affected by the available protocols between the machines. Our interface definition, called a *common network interface* in the thesis, provides a uniform set of services regardless of the actual connection-oriented transport protocols used to implement semantics of the interface.

## 1.2 Related work

This section describes the various approaches that have been taken to provide interoperability between network architectures or to provide a uniform interface to different protocols.

For processes to communicate across network architectures, protocol conversion has to take place to accommodate the differences between them. Some of the known solutions [8] provide interoperability between network architectures by

- defining useful partial resolution of protocol mismatches,
- complementing one or both protocols with missing services,
- or using an intermediary in the conversion process.

Protocol mismatches are classified by degrees of non-interoperability. Hard mismatch is a degree of mismatch that makes communication impossible. It results in situations like noncompletion of paths or deadlocks. Soft mismatch is an intermediate degree, while zero mismatch allows interoperability. Soft mismatches cause poor reliability in data transfer, missequenced data, buffer overflow and lower security. A probable cause for a soft mismatch is non-standard or incomplete implementations of the protocols. Protocol complementation is the provision of a set of functions that convert a hard mismatch between protocols to a soft mismatch. This enlarges the available set of protocol functions and makes it more interoperable. Use of an intermediary as a gateway reduces the number of solutions necessary for $K$ different protocols from $O(K^2)$ to $O(K)$. Conversion between a common subset of two different protocols provides some interoperability. Such an approach has been discussed by Groenbak [9] for protocols TCP and ISO Class 4.

The solutions discussed do not, however, alleviate the misery of a software developer using different interfaces to each protocol.

Auerbach [2] describes a toolkit called Transport Abstract Conversion Toolkit, or TACT, developed to provide conversion between, as he puts it, "services which the program *wants to use* and the set of services *actually available* between itself and the desired partner". TACT claims to provide true "transport abstraction conversion" by inserting protocol logic which was previously absent. It provides a set of small, modular, mutually compatible tools which can be combined to provide solutions to a larger problem. A TACT library is composed of

- a Transport Interface Package for every distinct network architecture. It is a set of functions which syntactically provides a interface called "extended sockets". It may be directly used by new applications for portability.

- a Programming Interface Package for every distinct programming interface. It translates the specifics of a programming language interface to the extended sockets interface.

- Abstraction converters to provide TACT abstraction of any protocol type.

AT&T's Transport Library Interface (TLI) is a uniform interface for the network application writers in Unix System V release 3. It provides access to various protocols that are supported by the TLI. The interface is uniform but the *services vary* depending upon the protocol invoked using the interface. Each transport provider, a protocol (e.g. TCP or OSI TP4), is required to adhere to a transport provider interface (TPI) that defines how the transport protocols interact with the TLI. The Unix kernel boundary separates the protocol modules that use the TPI from the user processes that use the TLI. TLI thus provides a uniform interface but different services.

The socket is the basic building block for interprocess communication (IPC) in the Berkeley version of Unix. The combination of a specific ar-

chitecture and a protocol defines a communication domain in the IPC implementation. Some of the domains suppbrted are Unix, Internet TCP, Internet UDP and Xerox XNS. Depending upon the domain used, the IPC services are different. The programmer makes a specific choice of the desired communication domain when requesting a socket[1] to be allocated. The implementation then associates a domain type with any further activity using this socket. This restricts the available services to the set provided by this selected domain. IPC thus provides a uniform programming interface but different services by associating a domain type to the socket.

## 1.3 Common network interface approach

The common network interface provides a *uniform interface* and a *uniform service* to all applications. It defines its own set of services that is sufficient to provide the basic goal of interprocess communication. It is similar to the extended socket interface provided in TACT for new applications. Whereas TACT also provides abstraction converters to convert every distinct interface into the extended socket interface, the common network interface does not have any provisions for automatically converting the existing interfaces to itself.

The common network interface is also useful when computing platforms support more than one protocol. The common network interface determines the common protocol(s) in a multiple-protocol environment, and provides the services defined by the common network interface using this (these) protocols. If no common protocol exists, attempts to establish communication fail. The common network interface thus provides *limited interoperability* because no protocol conversion or complementation is attempted. The interface masks the underlying specific protocols without greatly compromising reliability and efficiency. To support the claim, we will present performance results of

[1] an endpoint to which a name may be bound

an implementation of this interface design supporting TCP protocol from the Internet domain and the Class 4 protocol [16, 3] from the OSI domain.

The common network interface is designed as a package of subroutine calls or library that can be generated and configured for a particular operating environment. The application writer links this library with his or her code. The package is generated separately for each platform to reflect the different network architecture available. The implementation of the package takes care of the underlying protocols and provides the interface services by using underlying facilities judiciously.

# Chapter 2

# Common Network Interface Design

A brief overview of the existing interfaces is introduced in section 2.1. Section 2.2 discusses the issues involved in locating remote peers for the purpose of communication and synchronization. Section 2.3 describes the other services that an interface should provide for purpose of interprocess communication.

## 2.1 Existing models

There exist several models for interprocess communication in different communication domains.

- **Local Interprocess Communication.**
  For processes on the same machine, interprocess communication is provided by the use of
  - queues, semaphores or shared memory as in System V Unix and OS/2,
  - an interface, called IUCV, on VM/CMS that permits data exchange between VM address spaces,
  - pipes in Unix BSD version.

- **Remote Interprocess Communication**
  Such communication is provided to the user by interfaces supporting specific protocols depending on the network architecture available.
  - **Berkeley Sockets**
    This interface provides the following services depending upon the domain (network architecture) that is supported at the host site.
    * **Datagrams** are unreliable, connectionless delivery schemes using User Datagram Protocol (UDP) on the Internet architecture.
    * **Streams** are full-duplex connection-oriented reliable byte-stream delivery protocols using Transport Control Protocol (TCP) on the Internet architecture.
    * **Sequenced packets** are also full duplex and reliable. They are, however, packet sequenced. They are available in the Xerox Network Server (NS) domain.
  - **Conversation verbs**
    They are half-duplex, orderly and variable-length exchanges across machines. They can be byte mode or record mode. They are supported in SNA network architecture by its APPC (LU 6.2 Type) protocol [10].

## 2.2 Naming and binding

For a process to communicate with a remote one, the local process must have knowledge of the address of the remote process. The address is specific to the *domain*[1] which the remote process uses. It is a value that uniquely identifies the host machine and the process on that domain. This reference mechanism is architecture or protocol dependent. An interface definition that expects

[1] a network architecture and a protocol within that architecture

users to be aware of the domain desired requires this kind of address. When an interface provides a uniform service and the user is not aware of the actual domain being used by the service provider, a level of indirection has to be introduced by a mapping from the process to the address.

When two processes want to establish an association, one of the two has to initiate the association establishment procedure. Because of the asymmetry at the establishment phase, only the responder needs to advertise its existence. Choosing and advertising one's identity is a *naming* issue. The identity has to be unique to be unambiguously located by others. Mapping the identity to a physical location on the system is a *binding* issue. Naming and binding are tricky. Ideally the processes, using our common network interface, would like to communicate with their peer by *name*, e.g a system-wide unique identifier.

This introduces the need for a service that maps such a name to a machine/protocol specific address. It would typically be a *directory service* or a *nameservice* whose existence is universally known and is easily accessible. The common network interface assumes the existence of a nameservice and uses its capabilities to provide the naming and binding services. The services expected of the nameservice are

- to store a sequence of user defined values using a string as a key. The number and type of such values may vary.
- to retrieve the values associated with it using a string as a key.
- to delete or modify an entry specified by the key.
- to determine whether the key already exists as an entry.

This common network interface requires the user process or application to provide a symbolic name for that process. A symbolic name is best represented as an alphanumeric character string. This symbolic name serves as the key for other processes to reference this process. Representation of real

addresses of machines in the implementations of the protocols are hardly that simple. They can, however, be constructed from more meaningful attributes like a machine name or a session identifier. For example, the address on a TCP transport provider can be constructed from the Internet address of the machine and the port at which the process is running. Existing interfaces to TCP can convert these attributes to a TCP-specific address. Similarly the address in OSI can be transformed from the OSI-specific jargon to a string and vice versa. The mapping from the symbolic name to protocol specific addresses for the common network interface is simply an association of the symbolic name to a sequence of groups, one each for a supported domain. A group may contain one or more attributes for that domain.

In the present case, such domain specific attributes for the nameserver are represented by strings. Assuming the above services are available from the nameservice. Mappings can be established between the symbolic name and the attributes. The naming and binding are then transparent to the user. Since designing the naming and binding service is a significant work in itself, this implementation uses a system-wide accessible file to store the sequence of strings. The symbolic name of the process should be unique because the current implementation does not check whether the name is already in use.

In the future it could be possible to use a directory server like X.500 that is general enough to maintain any such mappings.

## 2.3 The common network interface features

A distributed application design involves communication between at least two peer entities. Communication between the peer entities may be connection-oriented or connectionless. A connectionless design adds the address information of the destination to every message exchanged. It involves no setup time and communication can be routed through different paths each time. Typically the order of the delivered messages is not guaranteed to be the

order of generation.

On the other hand, a connection-oriented protocol has a finite cost to set up an association between the peers. Once the association has been established data may be exchanged reliably without having any routing information attached to it. The ordering of the messages is preserved when delivered to the remote user. Our common network interface is *connection-oriented* with a *reliable* and *ordered* delivery scheme. Henceforth, an association to exchange information in this model will be called a *conversation.*

The conversation being modeled by the interface resembles a *data pipe* into which one process stuffs data at one end to be retrieved at the other end by its peer. The pipe has a finite volume and it is quite possible for the pipe to be filled up faster than it is emptied. Data could be delivered as a byte stream or segmented into packets or records. Byte mode is better than record-mode because it does not impede the development of any further layers using this interface. A record mode delivery imposes a format and is useful when layers adhere to a protocol which offers services other than basic data transfer. The common network interface promises sequenced data delivery and adopts *byte stream* as the network input and output model. There is no segmentation or concatenation at the common network interface level.

*Dialogue control* and *expedited flow* are two other protocol functions of this interface. These two features are not guaranteed by the interface; they serve as hints to the implementation. The interface uses these hints to decide which transport protocol options to be used to provide these services. If the functions cannot be possible with the available protocols, the interface uses default options. At the conversation establishment phase, the user provides the functions as desired characteristics of the conversation.

- *Dialogue control.* A conversation can be established to provide data flow in both directions, henceforth called *full-duplex*, or in one direction at a time, called *half-duplex*, or in one direction all the time, called *monologue.* The full-duplex connection is more costly to establish and

maintain. A full-duplex connection needs two separate paths, one in each direction. For each path, system buffers are allocated to maintain the incoming and outgoing data. Half-duplex requires each side to agree on whose turn it is to send data. This involves dialogue management calls. Monologue presupposes an irreversible agreement and cannot be changed subsequently. When communicating entities are simultaneously transmitting and receiving data the cost incurred in establishing a full-duplex connection could be worth it. Half-duplex is useful when data flows in one direction for some interval of time before switching direction. Then the overhead in exchanging turns is minimal. Monologue is useful for applications that log data without requiring synchronization or acknowledgements. Only options for full-duplex and half-duplex are provided. The default is full-duplex.

- *Expedited flow.* This allows some data to be marked as urgent. Such data may be delivered before the normal data already queued but always after any other queued urgent data. Delivery of only a small number of bytes is permitted. When a sender elects to transmit urgent data, an indication of such pending data in the internal buffers of the receiving host is passed on to the peer at this host. The default is no urgent delivery.

When a choice is provided at the interface level to the user processes, it is possible that the choices made by the cooperating processes do not match. On mismatches, the common network interface does not provide for further exchange of messages between processes to negotiate and reach a consensus. Instead, the default options are used.

*Multiplexing* of read and write on conversations is an important primitive for application developers. When an application maintains several conversations, it is not possible to read or write these conversations in a predetermined sequence. This is because of the asynchrony of events happening in a distributed environment. It may then be necessary for the application

to test conversations prior to any read or write activity on them to avoid unnecessary calls. If an event (e.g. arrival of data) is pending on the conversation, then an operation may be carried out. Write may not be possible on a conversation because the conversation buffers may be full (analogous to full pipe). Multiplexing services allow the program to select conversations on which activities are pending or are free to be used. Although a single call could select all the conversations ready for read or write, this interface has separate calls to select ready conversations for read and write. It is intended to simplify design of the application. An application user who may need to selectively disable conversations to multiplex, can use utility calls that mark a conversation unusable and then unmark it on demand.

Most primitives provide both *blocking* and *non-blocking* options. A blocking call returns only on complete success or failure (an error condition). A non-blocking call guarantees that the call will return. If some event was pending or the desired activity could be carried out, even partially, status is returned. A non-blocking call allows users to do other chores without waiting for an event that is yet to happen or is in progress. Although non-blocking calls are provided, multiplexing is useful to avoid the costs of launching a call when no activity is pending or can be carried out on it. Multiplexing allows a programmer to avoid making non-blocking or blocking calls on inactive or busy conversations. A programmer may even make a blocking call to synchronize with the remote process once the multiplexing services detect an activity on this conversation.

The common network interface is targeted to be used in a multi-thread or light-weight process environment. Specifically it has been used with the C Threads [6] package but extensions to other thread environments should be straightforward. This requires use of non-blocking semantics at the implementation level. Access to shared data must be mutually exclusive.

# Chapter 3

# Common Network Interface Calls

This chapter describes the interface library so that the users may use it in their programs. The parameters and the completion semantics of each interface call are described in detail. This chapter should serve as a reference for the users.

Of the two peers that establish a conversation[1], the one which initiates the process of establishment is called the *initiator* and the other is the *responder*. There is a notion of a name to be registered that creates an entry for the responding process entity in the nameserver. The responder needs to register itself to advertise its existence. It is up to the process to choose a string[2] which then becomes the "well known name" or symbolic name for this responder. The responder calls `config_rsrc` which creates an entry for the responder in the nameserver as well as allocating communication endpoints on the available protocols.

An initiator process establishes a conversation directly with a responder using the "well known name" of the responder. Once a responder and an initiator have established a conversation, a "conversation identifier" is returned to each of the callers involved. Now either the initiator or the responder may

[1] Refer to section 2.3 .

[2] A string, in C, is an array of characters with a null character (hex value 0) as the last one.

transmit data or receive data on this conversation using `ewrite` or `eread` calls respectively. The conversation is continued until either or both terminate the conversation normally or abnormally.

`OK` is returned if any call is successful. Otherwise the call returns an error code. Calls may be blocking or non-blocking. Blocking calls wait indefinitely until they complete with success or error. Non-blocking calls return immediately with success, error or status report. They may have to be retried later.

## 3.1 Creation, destruction, management

- `int config_rsrc(my_name,handle)`

  - `char *my_name; /* in parameter */`
    A string (name) used to register the calling entity with the name-server. The name must be unique[3].
  - `NA_rsrc *handle; /* out parameter */`
    A pointer to `NA_rsrc` is passed as a parameter. The handle is initialized and returned to the caller. The handle is subsequently used when the caller issues an accept request.

  The call returns `OK` if successful. It returns `NO_RESOURCE` if the user process has reached the maximum number of endpoints per process that the current implementation allows. It returns `CONFIG_FAILED` if an internal error occurred while completing this call. It returns `REPEAT_LATER` if C Threads was being used and a resource handle could not be obtained immediately.

[3]Currently the nameserver does not check to see whether a name is already registered. When two different users use the same name, the one who registers last masks out the other.

- `int delete_rsrc(handle)`
  Deallocates an interface endpoint and frees up allocated memory.
  - `NA_rsrc handle; /* in parameter */`
    The handle to the endpoint to be deallocated [4].

  Returns `OK`.

- `int eaccept(handle, blocking, conv_type, Cv_id, partner, sz_prtn)`
  Allows a responder to rendezvous with an initiator sometime in the future.
  - `NA_rsrc handle; /* in parameter */`
    The handle returned when the caller registers itself with the nameserver.
  - `int blocking; /* in parameter */`
    If this parameter is `BLOCKING`, the call blocks until it succeeds or has an error. If the parameter is `NON_BLOCKING` the call returns immediately.
  - `struct conv_char conv_type; /* in parameter */`
    where `struct conv_char` is defined as

```
     struct conv_char {
#define NI_HDX  0
#define NI_FDX  1
      int mode;/* HDX or FDX                          */
#define NI_NORMAL 0
#define NI_EXPEDITED 1
      int expedited;
```

[4] Implementation of this call is incomplete; it does not delete the symbolic name from the nameserver.

```
/* expedited message delivery allowed or not */
  };
```

The conversation characteristics are not guaranteed by the implementation. The characteristics are provided by the user as a hint to the implementation.

The option `mode` defines the type of exchange desired to be done on this conversation. If half-duplex is supported by specific transport protocols and the `mode` is `HDX` then protocols supporting half-duplex will be first tried to establish a conversation. The same is true for full-duplex. If the option is not available, any supported protocol will be used. This feature is not implemented and all conversations are full-duplex.

The option `expedited` allows for priority data transfer if defined as `NI_EXPEDITED`. As in the case of `mode` it is used only as a hint. If priority data transfer is sought, protocols supporting this feature will be tried first. If such a protocol is used to establish a conversation, then during the establishment phase this option is matched with partner's option. In case of a disagreement between them, the implementation does not allow for a negotiation and silently sets it to the default value. If, however, no such protocol can establish the conversation, the implementation uses the default option.

The default `mode` is full-duplex (`NI_FDX`). The default `expedited` is no urgent support (`NI_NORMAL`). The conversation characteristics are assumed to have been agreed upon by the communicating peers. Since it is not guaranteed, these characteristics should be used with care.

- `NA_conv *Cv_id; /* out parameter */`

  A pointer to `NA_conv`, this parameter is returned to be used subsequently to refer to this conversation for any activity on this

conversation.

- `char **partner; /* out parameter */`
  This parameter is filled up with the "name", a string, of the initiator of the conversation. The parameter should be a pointer to a pointer to a `char` because size of the string returned depends upon the actual string received. This parameter is not an entry in the nameserver. It is provided for use by the application-level processes and is not used for conversation management in any way.

- `int *sz_prtn; /* out parameter */`
  A pointer to an integer that specifies the size of the string in parameter `partner` if the call is successful.

`OK` is returned if the call is successful and a connection has been made. If the call fails, `ACCEPT_FAILED` is returned when some error occurred while the attempt was made. If `REPEAT_LATER` is returned, the caller should try again. A possible reason for `REPEAT_LATER` to be returned is when the internal timer expired and there is no pending request from any initiator. In the `BLOCKING` mode, the call does not return until a conversation is established or an error occurs. A `NON-BLOCKING` call returns if no initiator request is waiting.

- `int econnect(my_name, partner_name, blocking, conv_type, Cv_id)`
  Initiates a connection to a remote application to exchange information.

  - `char *my_name; /* in parameter */`
    A string which may be null. It reflects a "name" this application would like the responder to know it by. It is passed on to the remote peer entity. It does not affect the connection establishment in any way. It is not registered with the nameserver either.

  - `char *partner_name; /* in parameter */`

A string. This parameter is the "well known name" of the remote application to which the caller wants to connect. `partner_name` is used to look up in the nameserver to resolve the binding issue. This must be non-null.

- `int blocking; /* in parameter */`
  When `BLOCKING`, the call blocks until it succeeds or has an error. When `NON_BLOCKING`, the call returns immediately with success or a return code.
- `struct conv_char conv_type; /* in parameter */`
  As described in `eaccept`.
- `NA_conv *Cv_id; /* out parameter */`
  A pointer to `NA_conv`, this parameter is returned to be used subsequently to refer to this conversation for any activity on this conversation.

If the call is successful and a connection has been made, `OK` is returned. If the call fails and `CONNECT_FAILED` is returned, then some error occurred while attempt was made. If `REPEAT_LATER` is returned the caller should try again. `REPEAT_LATER` is returned when the internal timer expired or when it is likely the call may block and the option is non-blocking. If `IN_PROGRESS` is returned due to a `NON_BLOCKING` call, `retry_connect` should be used for confirming completion of this call. This status reflects that the call will complete in the near future with success or error.

- `retry_connect(Cv_id)`
  Checks the status of an `econnect` call that has been issued earlier as `NON_BLOCKING`.
  - `NA_conv Cv_id; /* in parameter */`
    Input parameter is the conversation to be checked for completion.

Returns `OK` if completed. If still in progress `IN_PROGRESS` is returned. `CONNECT_FAILED` is returned if an error occurred and the call could not complete normally.

- `int eclose(Cv_id)`
  Deallocates a conversation and frees up allocated memory.

  - `NA_conv Cv_id; /* in parameter */`
    The conversation to be deallocated.

  Returns `OK`.

- `int eabort(Cv_id)`
  Not provided yet.

## 3.2 Data transfer and multiplexing

- `ewrite(Cv_id, blocking, priority, data, data_len)`
  Call to transfer data to remote peer.

  - `NA_conv Cv_id; /* in parameter */`
    The parameter identifies the conversation on which the data is to be transmitted. This is the same identifier that was returned as a parameter during `eaccept` or `econnect` call.
  - `int blocking; /* in parameter */`
    If this parameter is `BLOCKING`, the call blocks until it succeeds or has an error. If it is `NON_BLOCKING`, the call returns immediately.
  - `int priority; /* in parameter */`
    If it is priority data, use `NI_EXPEDITED` otherwise use `NI_NORMAL`. Default is `NI_NORMAL`. The priority is not guaranteed and is tried *if and only if* `NI_EXPEDITED` was specified at conversation establishment phase.

- `char *data; /* in parameter */`

  A pointer to a buffer containing the data to be transmitted. When the call is successfully returned, the buffer may be reused.

- `int *data_len; /* in out parameter */`

  The size of the data to be transmitted on input. If the parameter `priority` is `NI_EXPEDITED`, only data up to a size `MAX_URG_SZ` can be written. On return, the parameter contains the value of the data actually transmitted if return code is `OK`.

If the call is successful and *some* data has been transferred, `OK` is returned. If the call fails and `WRITE_ERROR` is returned, then some error occurred while attempt was made. If `REPEAT_LATER` is returned, the caller should try again as no data could be transferred. If expedited transfer was attempted and disallowed, `NI_BAD_PRIORITY` is returned. If defined `BLOCKING`, the call returns when all data has been transferred or an error occurred. `NON_BLOCKING` makes a single or no attempt to transfer data depending on the condition of the conversation. The returned value may be equal to or less than size requested.

- `eread(Cv_id, blocking, priority, data, data_len)`

  Call to accept data from a remote peer entity.

  - `NA_conv Cv_id; /* in parameter */`

    The conversation on which the data is to be transmitted. This is the same identifier that was returned as a parameter during `eaccept` or `econnect` call.

  - `int blocking; /* in parameter */`

    If this parameter is `BLOCKING`, the call blocks until `data_len` bytes has been read or the conversation has been closed or an error has occurred. If it is `NON_BLOCKING` the call returns immediately.

  - `int priority; /* in parameter */`

priority should be NI_EXPEDITED, if NI_URG_PENDING was returned by the previous eread. Otherwise it should be NI_NORMAL. Default is NI_NORMAL.

– char *data; /* out parameter */
A pointer to a buffer where the data is read in. The buffer should have sufficient space allocated as specified by the next parameter. When the call is successfully returned, the buffer may be reused.

– int *data_len; /* in out parameter */
The size of the data to be read on input. Only data up to a size MAX_URG_SZ is read if priority is NI_EXPEDITED. The parameter on return contains the size of the data actually read if return code is OK.

If the call is successful and some data has been transferred, OK is returned. If the call is terminated because the conversation was terminated or broken, NI_EOT is returned. If NI_URG_PENDING is returned, urgent data is (still) pending and the next eread call should have NI_EXPEDITED priority. If NI_BAD_PRIORITY is returned, either normal read was attempted with urgent data pending or urgent read was attempted with no urgent data pending. If the call fails and READ_ERROR is returned, some error occurred while attempt was made. If REPEAT_LATER is returned, the caller should try again. BLOCKING call returns when the size of data requested has been read or an error occurred. NON_BLOCKING makes a single or no attempt to read data depending on the condition of the conversation. The returned value may be equal to or less than size requested.

- rselect(arr_of_conv, wait, max_elem, arr_selected)
Selects, within the specified time wait, conversations ready to be read among the conversations specified in the array arr_of_conv. Some elements of the array of conversation identifiers may be masked by using a

mark_conv call. Later these conversation identifiers may be unmasked for use by using the unmark_conv call. The mark_conv and unmark_conv calls are described in the Section 3.3 . The masked conversation identifiers are ignored.

- `NA_conv arr_of_conv[]; /* in parameter */`
  This parameter is an array of conversation identifiers. All these conversations are tested for ready read condition. Any improper conversation identifier is ignored.
- `int wait; /* in parameter */`
  Specifies the time in seconds for which the call waits and tests for each conversation identifier.
- `int max_elem; /* in out parameter */`
  On input, specifies the number of conversation identifiers in the arr_of_conv and arr_selected. On return, specifies the number of conversations selected.
- `int arr_selected[]; /* out parameter */`
  This array has a one-to-one correspondence with the arr_of_conv. For each ready conversation in arr_of_conv, the corresponding entry in arr_selected is 1. Ignored or non-selected conversations have a entry 0 in arr_selected.

Returns OK if one or more conversations have been selected. Returns TIMED_OUT if the timer ran out and no element was selected. Returns R_SELECT_FAILED if there is an error.

- `wselect(arr_of_conv, wait, max_elem, arr_selected)`
  Exactly like rselect except that it is used for writing and not for reading. It returns W_SELECT_FAILED if there is an error.

## 3.3 Utilities

- `error2str(error_code,error_str)`

  Converts numeric `error_code` to a more meaningful diagnostic message.

  - `int error_code; /* in parameter */`

    Error code returned by any of the foregoing function calls.

  - `char *error_str; /* out parameter */`

    Declare `error_str` as a character array of size `MAXERRSTRLEN` and pass the pointer to it as the parameter. On return contains a diagnostic string.

- `void mark_conv(Cv_id)`

  Marks the conversation identifier so that it is ignored for further activity on this conversation.

  - `NA_conv *Cv_id; /* in out parameter */`

    A pointer to the conversation identifier that is to be marked.

  If the parameter is already marked, the call does nothing. There is no return code.

- `void unmark_conv(Cv_id)`

  It unmarks the conversation identifier so that it may be used for further activity on this conversation.

  - `NA_conv *Cv_id; /* in out parameter */`

    A pointer to the conversation identifier that is to be unmarked.

  If the parameter is already unmarked, the call does nothing. There is no return code.

- `ni_init()`
  Available only if user is using C Threads package. If C Threads is being used, this call must be invoked before any other call to the common network interface library. No input or output parameters.

## 3.4 Directory service

This has not yet been implemented, and will depend on the design of the nameserver. It is assumed now that the application designer knows the symbolic names of the various cooperating processes.

# Chapter 4

# Results and Conclusions

Providing a new interface definition and using exisiting interfaces to implement a new one might degrade the performance severely and be less reliable. Comparative performance measurements were carried out using the native interfaces and the new interface. Results from the tests and inferences are discussed in section 4.1. Section 4.2 discusses the future work and the possible uses of common network interface.

## 4.1 Results

| Data Size | Common | | Existing | |
|---|---|---|---|---|
| | TCP | OSI | TCP | OSI |
| *bytes* | *seconds* | | *seconds* | |
| 100 | 0.22 | 0.24 | 0.05 | 0.21 |
| 1,000 | 0.23 | 0.28 | 0.06 | 0.21 |
| 10,000 | 0.23 | 0.42 | 0.13 | 0.22 |
| 100,000 | 1.01 | 1.45 | 0.86 | 1.23 |
| 1,000,000 | 8.75 | 10.63 | 8.81 | 9.07 |

Figure 4.1: Average timings

| Data Size | Best | | | | Worst | | | |
|---|---|---|---|---|---|---|---|---|
| | Common | | Existing | | Common | | Existing | |
| | TCP | OSI | TCP | OSI | TCP | OSI | TCP | OSI |
| *bytes* | *seconds* | | | | *seconds* | | | |
| 100 | 0.21 | 0.20 | 0.04 | 0.20 | 0.25 | 0.36 | 0.06 | 0.25 |
| 1,000 | 0.21 | 0.21 | 0.05 | 0.20 | 0.37 | 0.45 | 0.06 | 0.22 |
| 10,000 | 0.21 | 0.38 | 0.12 | 0.20 | 0.29 | 0.53 | 0.14 | 0.28 |
| 100,000 | 0.97 | 1.29 | 0.84 | 1.18 | 1.21 | 1.70 | 0.96 | 2.00 |
| 1,000,000 | 8.19 | 9.96 | 8.78 | 8.78 | 9.16 | 13.14 | 10.13 | 15.39 |

Figure 4.2: Best and worst timings

Test programs (stubs) were developed to establish a conversation, transfer a fixed amount of bytes in each direction and then close the conversation. They were coded using the interface calls and also the protocol specific calls directly. Each stub executed the above test a hundred times. The timings were obtained for message lengths in powers of ten. Twelve such readings were obtained per case. The best and the worst were deleted before the average was obtained. The test provides two results. It gives the average, the best and the worst case times for short conversation periods, typically seen in remote procedure call environments (See Fig. 4.1 and Fig. 4.2). It is also a measure of a certain amount of reliability. The repeated execution of the tests did not fail and gave expected results. The data transferred were in order and without any missing bytes.

The time taken to establish a conversation, transfer bytes in both directions and then close a conversation is no worse than four times the time taken by using native protocol calls. In fact the difference between the timings obtained using the interface and the native code differs only when the messages transferred in both directions are shorter than a few kilobytes.

The reasons for degradation of services by using this interface are

- use of the nameservice (only at the establishment phase),
- additional exchange between the two peers to transfer initiating partner's symbolic name (only at the establishment phase),
- the initialization of data objects within the implementation (only at the establishment phase),
- and referencing the data objects using the conversation identifier (during other phases).

When the messages are short, the data transfer time and the connection set-up time are comparable in the native case. Additional time is required for set-up when using the interface because of the reasons above. In most cases it is dictated by the time taken by the initiating side to resolve the address of the remote side by looking it up in the nameserver and to exchange setup information. Overheads due to the other two reasons are minimal and often not noticeable. This also explains the reason why for large amounts of data the timings for the interface and the native calls are nearly equal. A significant finding from the tests is that *the performance of the common network interface depends critically on the speed of retrieval from the nameserver using the symbolic name as the key.*

The above timings reflect only TCP and ISO stacks because they are supported on the Sun machines in the department. A "proof-of-concept" implementation has been designed (but not tested) for SNA on the OS/2 operating system.

## 4.2 Conclusions and future work

Implementation of the common network interface has been developed supporting TCP and OSI. The support of such an interface using APPC has been designed and the implementation is in progress. The implementation

does not degrade the performance severely. This has been discussed in the previous section. The implementation is also reliable because a messaging layer to provide the communication protocol for a remote procedure call (RPC) mechanism has been built successfully over this interface by others. This messaging layer provides additional specialized functions for the RPC mechanism using the interface calls. The two layers, messaging and RPC, are to be used as a tool for students in a first course in distributed systems to develop and learn the concepts of messaging and RPC.

A major limitation to the use of our interface for interoperability in any kind of network is the absence of protocol conversion. There must exist a common protocol between hosts for any meaningful exchange to take place. The static nature of generation of the package does not allow for portable applications at the binary level among platforms having different transport protocol support.

The common network interface is, however, a good run-time design to be used for developing distributed applications. Whenever the application has to be ported to a new platform with a transport protocol not currently supported, it can be added transparently to the application. This makes it a versatile support for distributed applications without greatly compromising any desirable functions and speed.

Certain improvements can be made in the current implementation.

- Currently if the initiating side issues a protocol-specific call to initiate a conversation which returns with a status indicating that the call is in progress, the implementation does not bother to try out the remaining supported protocols to determine whether a conversation can be established faster using them.

- As discussed above, a high performance nameserver is needed to improve the current implementation results.

- An array of integers is currently used to store and reference the identi-

fiers returned to the user. Since the implementation does not handle a large number of conversations or resource identifiers, a long word with bits representing used or unused identifiers should suffice. This is an implementation optimization for speedup.

- Further, the implementation should always guarantee an endpoint on each available protocol stack on the responder side. If an endpoint could not be allocated for a protocol when `config_rsrc` was called, but later became available, the endpoint should then be allocated and added to the interface endpoint details in the nameserver.

An extension of this implementation could provide protocol conversion. Assume host A supports protocol $P_x$, host B supports protocol $P_y$ and there is a third host C that is to provide the conversion by supporting both $P_x$ and $P_y$. The applications on A and B which wish to communicate with each other talk instead to special processes $S_a$ and $S_b$ on A and B respectively. These processes $S_a$ and $S_b$ route the exchange between A and B through a process $S_c$ running on C. The $S_c$ acts as an intermediary process that reads user data from $S_a$ and writes it back for $S_b$. Such a group of processes in tandem could provide a gateway and thus conversion.

# Bibliography


[1] AT&T.
*Network Programmer's Guide*, 1988.
UNIX System V/386.

[2] J. Auerbach.
A Protocol Conversion Toolkit.
IBM T.J Watson Research Center, Yorktown Heights, New York, November 1988.

[3] Transport Protocol Specification for Open Systems Interconnection (OSI) for CCITT Applications.
International Telegraph and Telephone Consultative Committee, October 1984.
Recommendation X.224.

[4] Transport Service Definition for Open Systems Interconnection (OSI) for CCITT Applications.
International Telegraph and Telephone Consultative Committee, October 1984.
Recommendation X.214.

[5] D.E. Comer.
*Internetworking with TCP/IP.*
Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1988.

[6] E.C. Cooper and R.P. Draves.

C threads.
CS 88-154, Carnegie Mellon University, 1988.

[7] Xerox Corporation.
*Internet Transport Protocols, Report XSIS 028112.*
Office Products Division, Network Systems Administration Office, 3333 Coyote Hill Road, Palo Alto, California, 1981.

[8] P.E. Green.
Protocol Conversion.
*IEEE Transactions on Communications*, COM-34(3):257–268, March 1986.

[9] I. Groenbaek.
Conversion between the TCP and ISO Transport Protocols as a method of achieving Interoperability between Data Communication Systems.
*IEEE Journal of Selected Areas in Communication*, SAC-4(2):288–296, February 1986.

[10] IBM.
*OS/2 Version 1.1 APPC Programming Reference*, 1988.

[11] Information Processing Systems — Open Systems Interconnection: Basic Reference Model.
International Organization for Standardization and International Electrotechnical Committee, 1984.
Interational Standard 7498.

[12] Information Processing Systems — Open Systems Interconnection: Service Conventions.
International Organization for Standardization and International Electrotechnical Committee, 1987.
Technical Report 8509.

[13] Jon B. Postel.
User Datagram Protocol.

Request for Comments 768, DDN Network Information Center, SRI International, August 1980.

[14] Jon B. Postel.
Internet Protocol.
Request for Comments 791, DDN Network Information Center, SRI International, September 1981.

[15] Jon B. Postel.
Transmission Control Protocol.
Request for Comments 793, DDN Network Information Center, SRI International, September 1981.

[16] M.T. Rose.
*The Open Book.*
Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1989.

# Appendix A

# Network Interface Implementation

This appendix describes the specific protocols supported in the current implementation, the data structures used and how each interface call provides the semantics described in Chapter 3 using these protocols. The network interface implementation supports two transport protocols. The protocols themselves support varying services which are described in Section A.1. Data structures to provide the network interface semantics using these protocols are described in Section A.2. They are called *data objects* throughout the entire document. The code outline for each call is provided in Section A.3 and is meant to be used as a reference for any further modifications on this implementation.

## A.1 Some connection-oriented protocols

### A.1.1 Transmission Control Protocol (TCP)

TCP [15, 5] is a communication protocol of the Internet protocol suite [14, 5] that provides reliable stream delivery. At the application level interface, the TCP provides the following features.

- *Stream Orientation.* The data is treated as a stream of bits, divided into 8-bit bytes or octets.

- *Virtual-Circuit Connection.* Protocol software modules, through mutual exchange of information, provide to the application a *connection* that emulates a dedicated hardware circuit, hence virtual-circuit.

- *Buffered Transfer.* The protocol software uses a different size for transmission efficiency than the size of data passed down by the application. So the software buffers data, rather than transmitting immediately.

- *Unstructured Stream.* This model does not honor record boundaries or any kind of structuring in the stream.

- *Full Duplex.* It provides simultaneous two way-data flow.

### A.1.2 OSI Transport Layer (TP4)

The ISO Transport Layer [16, 4, 3] closely parallels the TCP services. However, the TP4 differs from TCP in some features like:

- OSI Transport is *packet-oriented*, providing better buffer management when the user data is bigger than packet size. But user data buffers, smaller than a packet size, cannot be compressed into a single packet.

- OSI Transport does not provide for a graceful release. It is always *destructive.* Graceful release ensures all data buffered on the releasing side are transmitted before the connection is severed. The TCP also can selectively shut down conversations in a direction. When the release is destructive some data may be lost.

- It provides an *expedited* data delivery of *limited* size.

### A.1.3 APPC

APPC (SNA LU TYPE 6.2 support) [10] is the representative of the conversation type of communication model. It provides synchronous program-to-program communication with the following features.

- Flow of data in *one direction* with implicit change of direction if necessary.
- *Buffering* of data.
- Data can be transferred in *records* or *unstructured format.*
- *Synchronization* level of confirm. A request for confirm does not complete until the responding side has confirmed this request. All other calls in this model complete without any explicit acknowledgement from the other side.

Logical Unit (LU) is a communication product handling a set of atomic protocol functions. Before a conversation can be established between two processes, the respective LUs on each host supporting the processes must be connected to form a session. Conversations in the processes then use a session for their exchange of information using calls called conversation verbs.

## A.2 Data objects

This section describes the data-structures, called objects, used in the implementation. Several objects are used internally to maintain the protocol-specific details hidden from the user and to facilitate certain aspects of connection management and data transfer. They are globally defined and are active throughout the duration of the process.

- **Connection Resource.** This object has several fields, two for each protocol supported. The first field is the address detail of the protocol

and the second field indicates whether the first field has been initialized. The addresses, specific to each transport provider, are obtained when a user issues a `config_rsrc` or `connect` call to the interface and the Connection Resource is initialized to these addresses. The last field stores the well known name associated with the process. It is used later to delete the entry of this object in the name-service.

```
typedef struct conn_rsrc{
#ifdef TCP_SUPPORTED
    struct tcp_endpoint{
        struct sockaddr_in in_socket;
        int sock_id;
    } tcp_ep;               /* TCP address structure    */
#endif TCP_SUPPORTED
    unsigned short tcp_allocated;
                            /* 1 if TCP socket allocated */
#ifdef SNA_SUPPORTED
    struct  sna_endpoint{
        char    *lu_alias;
        char    *tp_name;
    } sna_ep;               /* SNA address structure    */
#endif TCP_SUPPORTED
    unsigned short sna_allocated;
                            /* 1 if SNA tp started      */
#ifdef ISO_SUPPORTED
    struct TSAPaddr *tsap_ep;
                            /*  OSI transport address ptr */
#endif ISO_SUPPORTED
    unsigned short osi_allocated;
                            /* 1 if OSI resource allocated*/
```

```
    char *well_known_name;
                        /* name with which it is      */
                        /* registered. needed while   */
                        /* deleting the entry from    */
                        /* name-server                */
} NA_conn_rsrc;
```

- **Connection Resource Table.** This is an array of pointers to Connection Resource objects that have been allocated. The *Connection Resource Identifier* returned to the user on the responding side indexes to this table to reference one of the Connection Resource objects that will be used for establishing a conversation. Currently it has a maximum size `MAX_CONVERSATIONS`.

```
NA_conn_rsrc *rsrc_arr[MAX_CONVERSATIONS];
```

- **Connection Resource Identifier.** This is an index to two tables – Resource Identifier Table and Connection Resource Table. It is returned to the user and identifies a particular communication endpoint allocation on one or more protocol suites.

- **Resource Identifier Table.** The table is used to search for the next unallocated index to be returned to the user as a Resource Identifier. The table is an array of integers of maximum size `MAX_CONVERSATIONS`. Resource identifiers returned range from 1 $\cdots$ `MAX_CONVERSATIONS`. Non-zero values in the table indicate that the indexes reference currently valid endpoints.

```
int ep_avail[MAX_CONVERSATIONS];
```

- **Conversation Object.** This object has nine fields. The first three are for a protocol-specific conversation handle returned by any of the stack that establishes the connection. Only one of them is valid at any time and this is reflected in the fourth field. The fifth and the sixth fields are relevant for non-blocking connects, as they store the status of the call in progress. The seventh field stores the priority, used later to send, accept or refuse priority data. The last two fields maintain the history of the kind of data sent or received over the conversation prior to this call.

```
typedef struct conv_info{
    int tcp_conv_id; /* TCP connection descriptor   */
    unsigned long sna_conv_id;
                     /* SNA connection descriptor   */
    int tsap_conv_id;/*  OSI transport layer        */
                     /*   conversation descriptor   */
    int network_type;/* = NI_TCP if TCP ,           */
                     /* = NI_SNA if SNA ,           */
                     /* = NI_OSI if OSI .           */
/*Only one kind of conversation may exist at a time */
    int connected;   /* = NI_DONE if non-blocking or*/
                     /* blocking call complete.     */
                     /* = NI_INPROGRESS if non-     */
                     /* blocking call is in progress*/
    char *my_name;   /* Caller's name for exchange  */
                     /* after initial establishment */
    int  priority;   /* priority type permitted     */
```

```
    int  recv_mode;  /* toggle between urgent and   */
                     /* normal receive mode         */
    int  send_mode;  /* toggle between urgent and   */
                     /* normal send mode            */
} NA_conv_info;
```

- **Conversation Identifier.** This is an index to two tables – *Conversation Table* and *Conversation Identifier Table.* It is returned to the user and references the conversation for any further activity desired on it.

- **Conversation Table.** This is an array of pointers to Conversation Objects. When a Conversation Object is initialized, the pointer to this object is stored in the entry indexed by the Conversation Identifier returned to the user. Currently the table has a maximum size of a predefined value `MAX_CONVERSATIONS`.

```
NA_conv_info *conv_arr[MAX_CONVERSATIONS];
```

- **Conversation Identifier Table.** The table is used to search for the next unallocated entry. The index of this entry is returned to the user as a Conversation Identifier. The table is an array of integers of maximum size `MAX_CONVERSATIONS`. Hence conversation identifiers returned range from 1 $\cdots$ `MAX_CONVERSATIONS`. Non-zero values in the table indicate that the indexes reference currently valid conversations.

```
int conv_avail[MAX_CONVERSATIONS];
```

- **Stream Object.** This emulates a read-only data stream of bytes. The object has three fields.

    1. Pointer to the stream.
    2. Size of the stream.
    3. Position from which the next read operation is effective.

  The pointer to the current read position advances only. Once the end has been reached, the object may be deallocated. It is used to convert packet sequenced data to stream type. It is used to maintain the sequenced packets that have been read internally but not by the user.

```
struct pkt2stream {
    char *data; /* the byte stream                        */
    int  size;  /* size of the byte stream                */
    int  curr_ptr;/* the current pointer to the          */
                 /* data unread at the user level        */
    };
/* this defines the structure to reference received */
/* data after converting from a buffer queue of OSI */
/* structure to stream type                         */
```

- **Stream Table.** This is an array of stream objects, one for each potential conversation.

```
struct pkt2stream recv_data_arr[MAX_CONVERSATIONS];
/* array of unread data as a stream              */
```

## A.3 Code outline

A conversation normally passes through three phases in its lifetime. The duration of each phase is totally application dependent.

- An establishment phase.
- A data transfer phase
- A release phase.

The *establishment phase* is asymmetric. The following calls, used in this phase, are implemented as described below. The parameters and the return codes are described in detail in Appendix A.

- `config_rsrc`
  1. Allocate an address on each transport provider supported by the implementation e.g. by issuing `sock` and `bind` calls on TCP or `TNetListen` in OSI.
  2. Initialize a Connection Resource and enter it in the Connection Resource Table for any future activity on it.
  3. Store the sequence of strings representing the allocated addresses indexed by the user supplied name in the name-server. The first two strings in the sequence represent the hostname and port of a TCP address in the Internet domain. The next string is the OSI transport address. The remaining two strings are the LU name and the TP name and constitute the SNA address. A string is left null if the corresponding transport provider is unable to allocate a communication endpoint or is not supported.
  4. Terminate with `OK` and a resource identifier if any endpoint is allocated and registered. Otherwise return appropriate error.

- `eaccept`

  1. Reference the Connection Resource Table for the proper Connection Resource object with the resource identifier passed.
  2. From the conversation characteristics, decide the order of polling among the protocols available which best fits the hints.
  3. Poll to check for any pending initiator request for every endpoint allocated in the Connection Resource object. This polling action is a non-blocking protocol-specific call on each allocated endpoint.
  4. Allocate and initialize a Conversation Object if a request is accepted internally by one of the endpoints.
  5. Get partner's symbolic name and exchange priority if required. In some protocols, priority may be a part of the establishment phase primitives. In others, the implementation may have to exchange messages to establish this.
  6. Update Conversation Identifier Table and Conversation Table.
  7. Return with a Conversation Identifier or error as appropriate.

- `econnect`

  1. Retrieve the sequence of strings that constitute the endpoint details from the name-server with the partner's symbolic name.
  2. From each non-null string, construct the addresses of the communication endpoints, i.e. the Connection Resource object of the peer.
  3. From the conversation characteristics, decide the order of polling among the protocols available which best fits the hints.
  4. Try asynchronously to connect to the peer with the protocol specific addresses. If the protocol-specific call to connect is success-

fully launched for that protocol stack then this call either completes or is in progress. If the interface call is blocking continue until protocol-specific call completes. If the interface call is non-blocking and the protocol specific call returns a status that it is in progress, set "connected" field of the Conversation Object to be `IN_PROGRESS` and return after other updates of the Conversation Object. A protocol-specific "initiate a conversation" call cannot block on that endpoint. So non-blocking protocol-specific calls are used to simulate blocking and non-blocking at the interface level.

5. Allocate and initialize a Conversation Object if the initiate call on an endpoint is accepted by the remote peer.

6. Transmit own symbolic name for user level interaction. Can be null if desired. Exchange priority if required. In some protocols, priority may be a part of the establishment phase primitives. In others, the implementation may have to exchange messages to match priorities.

7. Update Conversation Identifier Table and Conversation Table.

8. Return with a Conversation Identifier or error as appropriate.

- `retry_connect`

  1. Retrieve Conversation Object from the Conversation Object Table using the conversation identifier.

  2. Test for completion using calls specific to the active protocol.

  3. Update field "connected" of the Conversation Object.

  4. Transmit own symbolic name for user level interaction. Can be null if desired.

  5. Return with a Conversation Identifier or error as appropriate.

The *data transfer phase* uses the following calls.

- `eread`

  1. Retrieve the Conversation Object associated with the conversation from the Conversation Table with the Conversation Identifier.
  2. Issue appropriate calls to get data sent by the peer depending upon the protocol in use.
  3. Get data until exactly the size desired has been fetched if mode is blocking. Convert from records or packets to stream if required. If more data has been fetched than required by the interface user, allocate the remaining data in the Stream Object of the corresponding entry in the Stream Table indexed by the Conversation Identifier.
  4. Return with the data fetched by the protocol-specific call if the size is less than or equal to the data size requested by a non-blocking interface call. Fetch more data until desired size is read if the interface call is blocking. Allocate the remaining data in the Stream Object of the corresponding entry in the Stream Table indexed by the Conversation Identifier if the data read by protocol specific calls is more than requested by the user.

- `ewrite`

  1. Retrieve the Conversation Object associated with the conversation from the Conversation Table with the Conversation Identifier.
  2. Issue appropriate calls to send data to the peer depending upon the protocol in use.
  3. Send data until exactly the size desired has been sent even if it needs more than one protocol-specific call when the user requests non-blocking.
  4. Try one internal call to send as much data as possible if mode is non-blocking. Before sending the data, select to see if data can be

sent on the conversation. This will avoid blocking the call if the pipe is full.

- `read_mpx`

  1. Get unmarked conversations.
  2. Create the read masks for the conversations if the protocol is TCP or OSI.
  3. Issue the protocol-specific calls and return the selected conversations when they complete.

- `write_mpx`

  1. Get unmarked conversations.
  2. Create the write masks for the conversations if the protocol is TCP or OSI.
  3. Issue the protocol-specific calls and return the selected conversations when they complete.

The last phase is *connection release*. When a conversation is closed, the associated memory has to be deallocated and the tables have to reinitialized. A Communication Object may be deleted too. The calls are

- `eclose`

  1. Issue protocol-specific calls to close the conversation.
  2. Flush internal buffers if necessary.
  3. Deallocate the Conversation Object.
  4. Reset entries in the Conversation Table and the Conversation Identifier Table.

- `delete_rsrc`

1. Issue protocol-specific calls to delete the communication endpoints allocated on the protocol stacks.
2. Delete the entry in the name-server.
3. Deallocate the Address Object.
4. Reset entries in the Address Table and the Resource Identifier Table.

# Appendix B

# Compiling and Linking the Interface Calls

One include file and one library are necessary to use this code. The include file is `na.h` and contains all the necessary declarations and definitions. The library is generated as `libna.a`.

The following is a sample of a make-file which shows how the common network interface can be used. `server.c` is a source file using the common network interface and the executable file generated is `server`.

```
INCLUDE = ../include
OBJ = ../obj
LIB =  ../lib

CFLAGS=  -c -g -I$(INCLUDE)
LIBFLAGS = -L$(LIB)

server: server.o $(LIB)/libna.a
cc $(LIBFLAGS) -o server server.o -lna

server.o: server.c $(INCLUDE)/na.h
cc  $(CFLAGS) server.c
```

# Appendix C

# An Example Responder

This appendix lists an example responder code using the common network interface. The responder accepts conversations from an initiator, selects ready conversations and then reads data on these conversations. If the initiator has terminated the conversation, then the conversation is closed. The responder is in an infinite loop accepting requests for a conversation.

```
/************************************************************
*                       RESPONDER                           *
*                                                           *
*            Author Debashish Chatterjee                    *
*            Date   19th July 1990                          *
*                                                           *
*    This is an example of a process that accepts many     *
*    conversations and also selectively reads data from     *
*    them.                                                  *
************************************************************/

/*                    standard include file                */
#include "na.h"
```

```
/*  maximum number of conversations                                  */
#define max_elem    20

main(argc, argv)
int argc;
char *argv[];

{
    int ret_code,    /*  return code from interface calls    */
        len,         /* length of data                       */
        mode,        /*  blocking or non-blocking calls      */
        arr_of_sel[max_elem];   /*  array to reflect         */
                                /*selected conversations     */
    int i = 0;
    NA_rsrc handle; /*  handle to refer a configured         */
                    /*  resource                             */
    NA_conv a_conv, /*  a conversation identifier            */
            arr_of_conv[max_elem];  /*  array to keep        */
                                    /*  conversation         */
                                    /*  identifiers          */
    int wait;        /*  time in seconds to wait for the     */
                     /*  event to occurby the call           */
    char *partner;   /*  name of the peer                    */
    int sz_prtn,     /*  size in bytes of the partner        */
        sel_no,
        curr_conv = 0;
    int status = 0;
    int accepted = 0;
    char *buf;
    struct conv_char conv_type; /*  conversation             */
```

```
                                /*  characterstics                  */
int sz_2_rx;    /*  bytes to be received                            */

/*      command line arguments to the responder                     */
/*  adjusts some variable parameters in the code                    */
if (argc != 6)
{
    printf("Improper # of args.\n");
    printf("Specify <name> <secs> <mode> <urgent>");
    printf(" <buf size >\n");
    exit(0);
}

/* start of initialization                                          */
mode = atoi(argv[3]);
if (mode)
    mode = BLOCKING;
else
    mode = NON_BLOCKING;

/*  qualify the conversation characterstics                         */
conv_type.mode = NI_FDX;
conv_type.expedited = atoi(argv[4]);

sz_2_rx = atoi(argv[5]);
wait = atoi(argv[4]);

for (i = 0; i < max_elem; i++)
    arr_of_conv[i] = 0;
/*  end of initialization                                           */
```

```
/* register my EXUname with the nameserver                */
/* and use the returned  handle for accepting             */
/* a conversation on it                                   */
status = config_rsrc(argv[1], &handle);

if (status == OK)
/* EXUname properly registered                            */
{
    do
    {
        if (accepted < max_elem)
        /* not more than max-elem converstions            */
        {
            status = eaccept(handle, mode, conv_type,
                &a_conv, &partner, &sz_prtn);
            /* accept a conversation                      */
            if (status == OK)
            {
                printf("%d :partner exu is %s\n",
                                    a_conv, partner);
                /* print the returned                     */
                /* partner EXUname                        */
                for (i = 0; i < max_elem; i++)
                    if (arr_of_conv[i] == 0)
                    {
                    /* housekeeping                       */
                        arr_of_conv[i] = a_conv;
                        accepted++;
                        break;
```

```
            }
        }
    }
    if (accepted == 0)
        continue;   /*  no active conversation      */

    /* first select for read                        */
    sel_no = max_elem;
    ret_code = rselect(arr_of_conv, &sel_no,
                        arr_of_sel, wait);

    if (ret_code == R_SELECT_FAILED)
    /*  error while selecting                       */
    {
        printf("select returned error\n");
        exit(-1);
    }

    if (ret_code == OK)
    /*  some conversation is ready to be read       */
    {
        for (i = 0; i < max_elem; i++)
        {
            if (arr_of_sel[i] == 1)
            /* this is ready to be read             */
            {

                len = sz_2_rx;
                buf = (char *) malloc(len);
                if (errno == ENOMEM)
```

```
{
    printf("could not allocate");
    printf(" buffer\n");
    exit(-1);
}

ret_code = eread(arr_of_conv[i],
                BLOCKING,
                NI_NORMAL,
                buf, &len);
/* read data on the conversation     */
if (ret_code == OK)
{
    if (len > 0)
    {
    /* some data was read              */
        printf("#%d,size%d\n",
            arr_of_conv[i], len);
    }
}
else if (ret_code == NI_EOT)
{
/*  conversation has terminated      */
    eclose(arr_of_conv[i]);
    arr_of_conv[i] = 0;
    accepted--;
}
else
{
    printf("%d:read error:%d\n",
```

```
                                arr_of_conv[i], ret_code);
                               fflush(stdout);
                           }
                       }
                   }   /* for */
                   free(buf);
                   printf("done\n");
                   fflush(stdout);
               }
               /* else timed out so keep trying                     */

           } while (1);    /*  infinitely                           */
       }
       else
           printf("error on allocation %d\n", status);
   }
```

# Appendix D

# An Example Initiator

This appendix lists an example initiator code using the common network interface. The initiator requests a conversation from a responder and transfers some data before closing the conversation.

```
/************************************************************
*                    INITIATOR                              *
*                                                           *
*            Author Debashish Chatterjee                    *
*            Date   19th July 1990                          *
*                                                           *
*   This is an example of a process that initiates an       *
*   association with a remote responder. It transfers some  *
*   data several times and then terminate the association.  *
*                                                           *
************************************************************/

/*                  standard include files                 */
#include "na.h"
#include <sys/types.h>
#include <sys/time.h>
```

```
main(argc, argv)
int argc;
char *argv[];

{
    int i,
        j,
        ret_code,    /*  return code from interface calls    */
        urgent,      /*  priority level for data              */
        len,         /*  length of data                       */
        mode;        /*  blocking or non-blocking calls       */
    NA_conv a_conv; /*  a conversation identifier            */
    char *send_buf; /*  data to be sent                      */
    struct conv_char conv_type; /*  conversation             */
                                /*  characteristics          */
    int sz_2_s;     /*  number of bytes to send              */

    /*      command line arguments to the responder          */
    /*  adjusts some variable parameters in the code         */
    if (argc != 8)
    {
        printf("Improper # of args.\n Specify <partner>");
        printf(" <self> <delay> <mode> <urgent> <times> ")
        printf("<data size>\n");
        exit(0);
    }

    /* start of initialization                               */
    len = atoi(argv[7]);
```

```
sz_2_s = len;
send_buf = (char *) malloc(len);
if (errno == ENOMEM)
{
    printf("could not allocate buffer\n");
    exit(-1);
}

printf("start grind......\n");
for (i = 0; i < len; i++)
    send_buf[i] = '0' + (i % 10);
send_buf[i - 1] = '\0';
printf("Oh! what a relief!\n");

mode = atoi(argv[4]);
if (mode)
    mode = BLOCKING;
else
    mode = NON_BLOCKING;
j = atoi(argv[6]);
urgent = atoi(argv[5]); /*  in this example the priority*/
                        /*  is ignored                  */
conv_type.mode = NI_FDX;
conv_type.expedited = urgent;

do
    ret_code = econnect(argv[2], argv[1],
        mode, conv_type, &a_conv);
while (ret_code == REPEAT_LATER);
/* try to connect to the server                         */
```

```
    while ((ret_code == IN_PROGRESS) &&
        (mode == NON_BLOCKING))
        ret_code = retry_connect(a_conv);

    if (ret_code != OK)
    {
        printf("error on connect : %d\n", ret_code);
        exit(-1);
    }

    while (j--)
    {
    /* transfer data as many times as desired by the command*/
    /*  line argument <times>                               */

        len = sz_2_s;
        while ((i = ewrite(a_conv, mode, NI_NORMAL,
                        send_buf, &len)) == REPEAT_LATER);
        /* write data on the conversation                   */
        if (i != OK)
        {
            printf("error on write %d\n", i);
            break;
        }

#ifdef DEBUG
        printf("data size is %d\n", len);
        printf("done\n");
#endif DEBUG
```

```
        sleep(3);
    }
    eclose(a_conv);
}
```

# Appendix E

# Source Code Listing

## E.1 Include files and global declarations

```
/************************************************************
*                                                           *
*                       localconfig.h                       *
*                                                           *
*           Author Debashish Chatterjee                     *
*           Date   19th July 1990                           *
*                                                           *
*  Contains all the definitions that can be changed upon    *
*  choice of the generator of the library                   *
************************************************************/


#define WAIT_SEC 1
/* specify time in seconds the implementation has to wait   */
/* while selecting or multiplexing                          */
#define WAIT_U_SEC 1
/* specify time in micro seconds the implementation has to  */
/* wait while selecting or multiplexing                     */
```

```
#define LOCAL_TSAP_PORT_RANGE 10000
/* define this value if OSI available and 10000+ should be  */
/* used only by this library                                */
```

```
/************************************************************
*                                                           *
*                         na.h                              *
*                                                           *
*            Author Debashish Chatterjee                    *
*            Date   19th July 1990                          *
*                                                           *
*  Contains all the global structure definition             *
*  Data structure internal to implementation                *
************************************************************/

#ifndef _NA_CONFIG
#define _NA_CONFIG

#include "localconfig.h"
                /* defines the local configuration choices  */

/*              Unix related include files                  */
#include <stdio.h>
#include <signal.h>
#include <sys/file.h>
#include <sys/param.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/uio.h>
#include <sys/errno.h>
#include <sys/time.h>

/*              TCP related include files for sockets       */
#ifdef TCP_SUPPORTED
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#endif TCP_SUPPORTED



/*              ISODE include file                          */
#ifdef ISO_SUPPORTED
#include "tsap.h"
#endif ISO_SUPPORTED

/*             C THREADS include file                       */
#ifdef CTHREADS_SUPPORTED
#include "cthreads.h"
#endif CTHREADS_SUPPORTED

#ifndef OK
#define OK        0
#endif OK
#define TRUE      1
#define FALSE     0

/*  these are used internally should be moved to global.h   */
#define MAXDIGITS 6
#define MAX_CONVERSATIONS 20
#define MAX_EXUNAME 20
```

```
#define NONE     0
#define NI_OSI   2
#define NI_SNA   3
#define NI_TCP   4
#define OSI_SNA  5
#define OSI_TCP  6
#define SNA_TCP  7
#define ALL      9

#ifdef ISO_SUPPORTED
#define OSI_available 1
#endif
#ifndef ISO_SUPPORTED
#define OSI_available 0
#endif
#ifdef TCP_SUPPORTED
#define TCP_available 1
#endif
#ifndef TCP_SUPPORTED
#define TCP_available 0
#endif
#ifdef SNA_SUPPORTED
#define SNA_available 1
#endif
#ifndef SNA_SUPPORTED
#define SNA_available 0
#endif

/*          blocking options for users                      */
#define BLOCKING    1
```

```
#define NON_BLOCKING 0

/*          error codes                                      */
#define RSRC_ERROR -101 /* error while using protocol(WUP)  */
#define DATA_UNAVAILABLE  -102  /*  no data                  */
#define WRITE_ERROR  -103   /* internal error WUP            */
#define CONFIG_FAILED -104  /* error of nameservice or WUP   */
#define NO_RESOURCE -105        /*  all handles allocated    */
#define READ_ERROR  -106    /* internal error WUP            */
#define REPEAT_LATER -107   /* activity not possible now     */
#define ACCEPT_FAILED -108  /*  internal error WUP           */
#define CONNECT_FAILED -109 /*  internal error WUP           */
#define R_SELECT_FAILED -110 /* internal error WUP           */
#define W_SELECT_FAILED -111 /* internal error WUP           */
#define NO_SUCH_PARTNER -112 /* partner does not exist       */
#define TIMED_OUT -113      /*  no activity possible         */
#define BAD_C_HANDLE -114   /*  messed up conversation id    */
#define BAD_R_HANDLE -115   /*  messed up resource handle    */
#define IN_PROGRESS -116    /* connection in progress        */
#define NI_BAD_OPERATION -117   /* bad/inconsistent params   */
#define NI_EOT -100      /*  end of termination              */
#define NI_BAD_PRIORITY -118    /*  bad/disallowed priority */
#define NI_URG_PENDING -118 /* urgent data in queue          */

/*              urgent data size                             */
#ifdef ISO_SUPPORTED
#define MAX_URG_SZ TX_SIZE
#else
#define MAX_URG_SZ 16
#endif ISO_SUPPORTED
```

```
/*  connection resource handle and conversation identifier  */
typedef int NA_rsrc;
typedef int NA_conv;

/*  conversation characteristics                             */
struct conv_char {
#define NI_HDX  0
#define NI_FDX  1
    int mode; /* Monologue, half duplex or full duplex      */
#define NI_NORMAL 0
#define NI_EXPEDITED 1
    int expedited;  /* expedited message delivery allowed?  */
};

#endif  _NA_CONFIG
```

```
/***********************************************************
*                                                          *
*                    global.h                              *
*                                                          *
*           Author Debashish Chatterjee                    *
*           Date   19th July 1990                          *
*                                                          *
*  Contains all the global structure definition            *
*  Data structure internal to implementation               *
***********************************************************/

#include "na.h" /* standard declaration file               */

#ifdef ISO_SUPPORTED
#define TSAP_IN_PORT LOCAL_TSAP_PORT_RANGE
/* start allocating internet port for ISODE emulation of TP */
/*  layer from this                                         */
#endif ISO_SUPPORTED

#define NI_DONE 1
#define NI_INPROGRESS 2

char            *malloc();

typedef struct conn_rsrc{
#ifdef TCP_SUPPORTED
    struct tcp_endpoint{
        struct sockaddr_in in_socket;
        int sock_id;
    } tcp_ep;   /*          TCP allocation structure        */
```

```
#endif TCP_SUPPORTED
    unsigned short tcp_allocated;/* 1 if TCP allocated       */
#ifdef SNA_SUPPORTED
    struct  sna_endpoint{
        char    tp_id[8];
        char    tp_name[64];
    } sna_ep; /*        SNA allocation structure             */
#endif SNA_SUPPORTED
    unsigned short sna_allocated;   /* 1 if SNA tp started  */
#ifdef ISO_SUPPORTED
    struct TSAPaddr *tsap_ep;
/*  OSI transport access point address ptr                   */
#endif ISO_SUPPORTED
    unsigned short osi_allocated;
                              /* 1 if OSI resource allocated  */
    char *well_known_name;
                          /* name with which it is registered */
                          /* needed while deleting resource   */
} NA_conn_rsrc;
/* The above structure is the reference structure to the     */
/* various endpoints established for different suites. It is*/
/* used while establishing a conversation between processes.*/
/* It cannot be a union as it is possible for a machine to  */
/* support more than one network architecture.              */

typedef struct conv_info{
    int tcp_conv_id;            /* TCP connection descriptor*/
    unsigned long sna_conv_id;  /* SNA conversation id      */
    int tsap_conv_id;           /*  OSI conversation id     */
    int network_type; /* = NI_TCP if TCP , = NI_SNA if SNA ,*/
```

```
                     /*   = NI_OSI if OSI . Only one kind  */
                     /* of conversation may exist at any   */
                     /* time                               */
    int connected;  /* NI_DONE if actually connected else  */
                    /* NI_INPROGRESS if still in progress  */
    char *my_name;  /* initiator's name for exchange while */
                    /* connecting                          */
    int  priority;  /* priority data send or read supported?*/
    int  recv_mode; /* toggle between urgent and normal    */
                    /* receive mode                        */
    int  send_mode; /* toggle between urgent and normal    */
                    /* send mode                           */
} NA_conv_info;
/* This contains the details of the physical connection    */

struct osi_stream {
    char *data;     /*      the byte stream                */
    int  size;      /*      size of the byte stream        */
    int  curr_ptr;  /* the current pointer to the data     */
                    /* unread at the user level            */
};
/* this defines the structure to reference received data   */
/* after converting from a packet queue of OSI structure   */

#ifdef CTHREADS_SUPPORTED
mutex_t na_rsrc_lock,/*  to lock resource identifiers while */
                     /*  looking for a free one             */
        na_conv_lock;/*  to lock conversation identifiers   */
                     /*  while looking for a free one       */
#endif  CTHREADS_SUPPORTED
```

```
/*************************************************************
*                                                            *
*                    shared.h                                *
*                                                            *
*           Author Debashish Chatterjee                      *
*           Date   19th July 1990                            *
*                                                            *
*  Contains all the global structure declarations to be      *
*  shared by all modules                                     *
*  Data structure internal to implementation                 *
*************************************************************/
extern int ep_avail[MAX_CONVERSATIONS];
/*      array of assigned and unassigned resource handle    */
extern int conv_avail[MAX_CONVERSATIONS];
/*array of assigned and unassigned conversation identifiers */

extern NA_conn_rsrc *rsrc_arr[MAX_CONVERSATIONS];
/* array of pointers to resource structure                  */

extern NA_conv_info *conv_arr[MAX_CONVERSATIONS];
/* array of pointers to conversation structure              */

extern struct osi_stream recv_data_arr[MAX_CONVERSATIONS];
/* array of data as a stream unread by the user application */
```

```
/**************************************************************
*                                                             *
*           names.h                                           *
*                                                             *
*      Donald L. Stone                                        *
*   nameservice include file                                  *
**************************************************************/

extern short init_names();
extern short register_name();
extern short lookup_name();
```

```
/************************************************************
*                      global.c                             *
*                                                           *
*           Author Debashish Chatterjee                     *
*           Date   19th July 1990                           *
*   Contains all the global data structures for the library *
************************************************************/

#include "global.h"     /* data structure declaration  file */


int ep_avail[MAX_CONVERSATIONS];
/* array of assigned and unassigned resource handle         */

int conv_avail[MAX_CONVERSATIONS];
/* array of assigned and unassigned conversation cookies    */

NA_conn_rsrc *rsrc_arr[MAX_CONVERSATIONS];
/* array of pointers to resource structure                  */

NA_conv_info *conv_arr[MAX_CONVERSATIONS];
/* array of pointers to conversation structure              */

struct osi_stream recv_data_arr[MAX_CONVERSATIONS];
/* array of data as a stream unread by the user application */
```

## E.2 Creation, destruction, management

```
/**********************************************************
*                                                         *
*                     configure.c                         *
*                                                         *
*            Author Debashish Chatterjee                  *
*            Date   19th July 1990                        *
*                                                         *
*  Contains all the configuration related routines.      *
*  The routines are config_rsrc and rtrv_prtnr_rsrc.      *
**********************************************************/

/*           define the include files                    */
#include "global.h"
#include "names.h"
#include "shared.h"

#define MAXSTRLEN 132
/* maximum expected length when OSI specific protocol is   */
/* expressed as a string                                   */

struct hostent *lookup(name)
/* return the real address of the machine "name"           */
char *name;
{
    struct hostent *gethostbyname();
    struct hostent *address;

    address = gethostbyname(name);
    return address;
```

```
}


int register_rsrc(my_name, ep_info)
/* registers the allocated protocol endpoints as          */
/* strings in the name-server                             */
/* uses a basic implementation of the name service        */
/* maybe some day a high performance name service         */
NA_conn_rsrc *ep_info;      /* the protocol endpoints     */
char *my_name;              /* the key for the name-server */
                            /* also the symbolic name     */
{
    int i,
        len,
        fd,
        offset;
    char *values[5];

    /* the array of strings associated with the well      */
    /* known name                                         */
    /* in the nameserver. 2 for TCP, 1 for OSI, 2 for SNA */
    /* in that order                                      */

    values[0] = malloc(MAXHOSTNAMELEN); /* TCP hostname   */
    values[1] = malloc(MAXDIGITS);      /* TCP portnumber */

    values[2] = malloc(MAXSTRLEN);      /* OSI address as */
                                        /* string         */

    values[3] = malloc(MAXSTRLEN);  /* SNA tp name        */
```

```
    values[4] = malloc(MAXSTRLEN);  /* SNA LU Name          */

    for (i = 0; i < 5; i++)
        strcpy(values[i], "");

#ifdef TCP_SUPPORTED
    if (ep_info->tcp_allocated)
    {
        /* port # allocated by host to be extracted         */
        len = sizeof(ep_info->tcp_ep.in_socket);
        if (getsockname(ep_info->tcp_ep.sock_id,
                &(ep_info->tcp_ep.in_socket), &len) == -1)
        {

#ifdef DEBUG
            perror("getting socket name");
#endif DEBUG

            close(ep_info->tcp_ep.sock_id);
            ep_info->tcp_allocated = 0;
        }
        else
        {

            if (gethostname(values[0], MAXHOSTNAMELEN) < 0)
            {

#ifdef DEBUG
                perror("getting current hostname");
#endif DEBUG
```

```
                close(ep_info->tcp_ep.sock_id);
                ep_info->tcp_allocated = 0;
            }
            else
                /* both calls succeed                                */
                sprintf(values[1], "%d",
                    ntohs(ep_info->tcp_ep.in_socket.
                        sin_port));
        }
    }
#endif TCP_SUPPORTED

#ifdef ISO_SUPPORTED
    if (ep_info->osi_allocated)
    {
        /* convert transport address to string for           */
        /* registering                                       */
        values[2] = taddr2str(ep_info->tsap_ep);

#ifdef DEBUG
        printf("TSAP address is %s\n", values[2]);
#endif DEBUG
    }
#endif ISO_SUPPORTED

    if (ep_info->tcp_allocated || ep_info->osi_allocated ||
        ep_info->sna_allocated)
    {
        if (register_name(my_name, values, 5) < 0)
```

```
        /* call to the name-server to register the sequence */
        {

#ifdef DEBUG
            printf("register error \n");
#endif DEBUG

            return (-1);
        }
        else
            return (0);
    }
    else
        return (-1);
}
```

```
/**********************************************************
*                                                         *
*   int rtrv_prtnr_rsrc(partner_name,ep_info)             *
*   Input:  "partner_name" is the symbolic name to be used   *
*           as the key to retrieve the addresses          *
*   Output: "ep_info" is the communication resource endpoint *
*                                                         *
*   Returns 0 if successful else returns -1               *
*   This is used internally by econnect                   *
*                                                         *
**********************************************************/

int rtrv_prtnr_rsrc(partner_name, ep_info)
NA_conn_rsrc *ep_info;
char *partner_name;
{
    int len,
        fd,
        offset;
    char *my_name;
    struct hostent *address;
    char *values[5];

    /* an array of strings associated with my_name in the   */
    /* nameserver. 2 for TCP, 1 for OSI, 2 for SNA in       */
    /* that order                                           */

    values[0] = malloc(MAXHOSTNAMELEN);
    values[1] = malloc(MAXDIGITS);
```

```
    if (lookup_name(partner_name, values, 5) < 0)
    /* call to name-server to retreive the sequence          */
    {

#ifdef DEBUG
        printf("read error on directory lookup \n");
#endif DEBUG

        return (-1);
    }

#ifdef TCP_SUPPORTED
    if ((strcmp(values[0], "") == 0) &&
        (strcmp(values[1], "") == 0))
        /* No TCP protocol available                         */
        ep_info->tcp_allocated = 0;
    else
    {
        address = lookup(values[0]);
        bcopy((char *) address->h_addr,
            (char *) &(ep_info->tcp_ep.in_socket.sin_addr),
            address->h_length);
        ep_info->tcp_ep.in_socket.sin_port =
            htons(atoi(values[1]));
        ep_info->tcp_allocated = 1;
    }
#endif TCP_SUPPORTED

#ifdef ISO_SUPPORTED
    ep_info->osi_allocated = 0;
```

```
    /* Convert string to transport address if non-null      */
    if ((strcmp(values[2], "") != 0))
    {
        if ((ep_info->tsap_ep =
                str2taddr(values[2])) != NULLTA)
            ep_info->osi_allocated = 1;
    }
#endif ISO_SUPPORTED

    if (ep_info->tcp_allocated || ep_info->osi_allocated ||
        ep_info->sna_allocated)
        /* At least one protocol exists between them        */
        return (0);

    return (-1);
}
```

```
/***********************************************************
*                                                          *
*   int config_rsrc(my_name,handle)                        *
*   Input:  "my_name" is the symbolic name it wishes to be *
*           known in the world.                            *
*   Output: "handle" is a nonnegative integer returned if  *
*           call returns OK.                               *
*   Returns OK if successful else returns negative integer *
*   denoting CONFIG_FAILED or NO_RESOURCE.                 *
*                                                          *
***********************************************************/
int config_rsrc(my_name, handle)

char *my_name;

/* The symbolic name which uniquely identifies the process */

NA_rsrc *handle;

/* this is the information to be returned for              */
/* accepting future conversations                          */
{
    struct hostent *in_mc_info; /* TCP specific host  info */
    int s,
        i;          /* local variables                     */
    NA_conn_rsrc *ep_info;  /* communication resource      */
                            /* object                      */

    /* find a free slot in the data my data structure      */
#ifdef CTHREADS_SUPPORTED
```

```
    if (mutex_try_lock(na_rsrc_lock) == FALSE)
        return REPEAT_LATER;
#endif CTHREADS_SUPPORTED

    for (i = 0; i < MAX_CONVERSATIONS; i++)
        if (!ep_avail[i])
            break;

#ifdef CTHREADS_SUPPORTED
    mutex_unlock(na_rsrc_lock);
#endif CTHREADS_SUPPORTED

    if (i == MAX_CONVERSATIONS)
        return NO_RESOURCE;
    ep_avail[i] = 1;
    (*handle) = i + 1;  /* handle ranges from                  */
                        /* 1..MAX_CONVERSATIONS                */

    ep_info = (NA_conn_rsrc *) malloc
        ((unsigned) sizeof(NA_conn_rsrc));
    ep_info->tcp_allocated = 0;
    ep_info->sna_allocated = 0;
    ep_info->osi_allocated = 0;


#ifdef TCP_SUPPORTED
    if (TCP_available)
    {

        s = socket(AF_INET, SOCK_STREAM, 0);
```

```
        if (s > 0)  /* only if socket allocated deleted      */
                    /* return NO_RESOURCE                    */

            ep_info->tcp_ep.sock_id = s;
        ep_info->tcp_ep.in_socket.sin_family = AF_INET;
        ep_info->tcp_ep.in_socket.sin_addr.s_addr =
            INADDR_ANY;
        ep_info->tcp_ep.in_socket.sin_port = 0;

        if (bind(s,
                (struct sockaddr *)
                & (ep_info->tcp_ep.in_socket),
                sizeof(ep_info->tcp_ep.in_socket))
            == -1)
        {

#ifdef DEBUG
            perror("bind:");
#endif DEBUG

            ep_info->tcp_allocated = 0;
            close(s);
        }
        else
        {
            /* startup listen for TCP                        */

            if (listen(ep_info->tcp_ep.sock_id,
                        5) == -1)
            /* hacked to 5 ; it is the BSD4.2 limit          */
```

```
            {
                close(ep_info->tcp_ep.sock_id);
                ep_info->tcp_allocated = 0;
            }
            else
                ep_info->tcp_allocated = 1;
        }

    }

#endif TCP_SUPPORTED

#ifdef SNA_SUPPORTED
    if (SNA_available)
    {
        /* do something                                              */
    }

#endif SNA_SUPPORTED

#ifdef ISO_SUPPORTED
    if (OSI_available)
    {
        char buffer[BUFSIZ];
        int tsap_in_port;
        struct TSAPdisconnect *td;

        isodetailor(my_name, 1);
        tsap_in_port = TSAP_IN_PORT;
        /* defined in the include file,it is needed because */
```

```
        /* ISODE emulates OSI over TCP and the port numbers */
        /* are to be provide explicitly                     */
        td = (struct TSAPdisconnect *)
            malloc(sizeof(struct TSAPdisconnect));

        do
        {
            (void) sprintf(buffer, "Internet=%s+%d",
                TLocalHostName(), tsap_in_port);
            /* convert internet address to OSI transport    */
            if ((ep_info->tsap_ep = str2taddr(buffer))
                == NULLTA)
            {

#ifdef DEBUG
                printf("error in address translation \n");
#endif DEBUG

                ep_info->osi_allocated = 0;
                break;
            }

            if (TNetListen(ep_info->tsap_ep, td) == NOTOK)
            {

#ifdef DEBUG
                printf("TSAP error in listen, %s\n",
                    TErrString(td->td_reason));
#endif DEBUG
```

```
                if (td->td_reason == DR_CONGEST)
                /* internet port already in use                     */
                {
                    tsap_in_port++;
                    free(ep_info->tsap_ep);
                    continue;
                }
            }
            else
            {

#ifdef DEBUG
                printf(" TSAP listen done successfully\n");
#endif DEBUG

                ep_info->osi_allocated = 1;
                break;
            }

        } while (TRUE);
    }

#endif ISO_SUPPORTED

    if (ep_info->tcp_allocated || ep_info->sna_allocated ||
        ep_info->osi_allocated)
    {
    /* at least one communication endpoint has been           */
    /* allocated                                              */
        if (register_rsrc(my_name, ep_info) < 0)
```

```
        {
            (*handle) = -1;
            free(ep_info);
            return CONFIG_FAILED;
        }


    }

    if (ep_info->tcp_allocated || ep_info->sna_allocated ||
        ep_info->osi_allocated)
    {
        rsrc_arr[*handle - 1] = ep_info;
        return OK;
    }

    (*handle) = -1;
    free(ep_info);
    return CONFIG_FAILED;
    /* for lack of a communication endpoint                   */
}
```

```
/***********************************************************
*                                                          *
*                    eaccept.c                             *
*                                                          *
*          Author Debashish Chatterjee                     *
*          Date   19th July 1990                           *
*                                                          *
*   This is the implementation of the call "eaccept"       *
*                                                          *
*   int eaccept(handle, blocking, conv_type, Cv_id, partner, *
*                                                 sz_prtn) *
*   "eaccept" accepts a connection from a remote process   *
*   in a blocking or  non-blocking mode.                   *
*   Input:  The "handle" returned when the communication   *
*           endpoint has been allocated.                   *
*           Blocking or non-blocking mode "blocking"       *
*           Kind of conversation by the server and client  *
*           using "conv_type"                              *
*   Output : The conversation identifier "Cv_id" of        *
*           type NA_conv                                   *
*           The symbolic name "partner" of the partner     *
*           it accepted from.                              *
*           "sz_prtn" is the size of the partner name      *
*   Returns OK if successful. Else  ACCEPT_FAILED if failed *
*   while trying to accept due to invalid partner name or  *
*   non-existing partner.                                  *
*   If maximum allowable resources have been used up       *
*   NO_RESOURCE is returned.                               *
*   If the mode is non-blocking and the call could not be  *
*   launched REPEAT_LATER is returned. The caller should   *
```

```
*   try again.                                               *
*                                                            *
*************************************************************/

/*  include files from the include directory                */
#include "global.h"
#include "shared.h"
```

```
/**************************************************************
*                                                             *
*       This function gets the initiator's name once the      *
*       the association has been established                  *
*       Returns the partner's name and length of the name     *
*       If there is an error, the size is -1                  *
*                                                             *
**************************************************************/
char *get_prtnr_name(Cv_id, status)
NA_conv Cv_id;  /*  conversation identifier                  */
int *status;    /*  size of the string returned              */
{
   int sz_prtn;    /* size of the partner name                */
   int j,
       bytes_2_read;   /* bytes to be read                    */
   char *partner;  /* partner's name is filled up here        */


   /* read an integer  telling how many bytes of partner      */
   /* name to be read next                                    */
   j = sizeof(int);
   if (eread(Cv_id, BLOCKING, NI_NORMAL, &sz_prtn, &j) == OK)
   {
       if (j == sizeof(int))
       {

#ifdef DEBUG
            printf("eaccept : partner length %d\n", sz_prtn);
#endif DEBUG
```

```
            /*          received the correct size            */
            bytes_2_read = sz_prtn;
            partner = (char *) malloc(sz_prtn);
            if (eread(Cv_id, BLOCKING, NI_NORMAL,
                                    partner, &sz_prtn) == OK)
            {
                if (sz_prtn == bytes_2_read)
                {
                    *status = sz_prtn;
                    return partner;
                }

#ifdef DEBUG
    printf("eaccept : partner length not as should be\n");
#endif DEBUG
            }

#ifdef DEBUG
    printf("eaccept : size of partner name not received \n");
#endif DEBUG
        }
    }
    *status = -1;
    return ((char *) 0);
}
```

```
#ifdef TCP_SUPPORTED
/***********************************************************
*                                                          *
*       This function exchanges with the initiator the     *
*       priority once the association has been established *
*       This is required only for TCP protocol.            *
*       Sets the priority of the conversation apppropriately*
*       If there is an error, -1 is returned               *
*       If there is a match 1 is returned else 0           *
*                                                          *
***********************************************************/
int recv_send_priority(Cv_id, priority)
NA_conv Cv_id;
int priority;
{
    int j,
        prtnr_pr;

    j = sizeof(int);
    /* receive prtnr's priority                              */
    if (eread(Cv_id, BLOCKING, NI_NORMAL,
                                &prtnr_pr, &j) == OK)
    {
        if (j == sizeof(int))
        {

#ifdef DEBUG
            printf("eaccept : partner's priority is %d\n",
                                            prtnr_pr);
#endif DEBUG
```

```
            /*          send own priority                       */
            if (ewrite(Cv_id, BLOCKING, NI_NORMAL,
                                        &priority, &j) == OK)
            {
                if (j == sizeof(int))
                {
                    if (priority == prtnr_pr)
                        return 1;   /*     both agree           */
                    return 0;   /*      mismatch                */
                }
            }
        }

    }
    return -1;      /*          error in protocol               */
}

#endif TCP_SUPPORTED
```

```
/************************************************************
*                    the eaccept call                       *
************************************************************/
int eaccept(handle, blocking, conv_type, Cv_id,
                                        partner, sz_prtn)
NA_rsrc handle; /* communication resource object identifier */
int blocking;   /* the mode i.e  BLOCKING or NON-BLOCKING   */
struct conv_char conv_type; /* conversation characteristics */
NA_conv *Cv_id; /* the identifier to be returned to the user*/
                /* to reference the conversation once done  */
char **partner; /* the partner name to be filled up         */
int *sz_prtn;   /* the size of the partner's name returned  */

{
    /* Find out on which networks communication endpoint    */
    /* have been allocated                                  */

    /* If blocking, return only if a connect is requested   */
    /* from remote site or error occurred                   */
    /* If non-blocking, return immediately if no pending    */
    /* connects                                             */

#ifdef TCP_SUPPORTED
    /*          TCP specific declarations                   */
    struct sockaddr_in *partner_addr;
    fd_set read_template;

#endif TCP_SUPPORTED

#ifdef ISO_SUPPORTED
```

```
    /*          OSI specific declarations                          */
    struct TSAPstart *ts;
    struct TSAPdisconnect *td;

#endif ISO_SUPPORTED
    struct timeval wait;
    int len;
    int status,
        curr_conv_index;
    NA_conv_info *conversation; /*  a conversation object   */
    NA_conn_rsrc *ep_info;  /*  the communication resource  */
    int tcp_urg,
        osi_urg,
        sna_urg;

    if ((handle < 1) || (handle > MAX_CONVERSATIONS))
        return BAD_R_HANDLE;
    /* communication endpoint identifier invalid range       */
    handle--;   /* handle is decremented as table indexes    */
                /* are from zero to MAX_CONVERSATION - 1     */
    if (ep_avail[handle] == 0)
        return BAD_R_HANDLE;
    /* the handle does not reflect an allocated resource     */

#ifdef CTHREADS_SUPPORTED
    if (blocking == BLOCKING)
        mutex_lock(na_conv_lock);
        /*      blocks thread till success                   */
    else
    {
```

```
        if (mutex_try_lock(na_conv_lock) == FALSE)
            return REPEAT_LATER;
    }
#endif CTHREADS_SUPPORTED

    for (curr_conv_index = 0;
            curr_conv_index < MAX_CONVERSATIONS;
                                    curr_conv_index++)
        if (!conv_avail[curr_conv_index])
            break;

#ifdef CTHREADS_SUPPORTED
    mutex_unlock(na_conv_lock);
#endif CTHREADS_SUPPORTED

    if (curr_conv_index == MAX_CONVERSATIONS)
        return NO_RESOURCE;

    conversation = (NA_conv_info *)
                    malloc((unsigned) sizeof(NA_conv_info));

    /*                      Initialize                      */

#ifdef ISO_SUPPORTED
    ts = (struct TSAPstart *)
                    malloc(sizeof(struct TSAPstart));
    td = (struct TSAPdisconnect *)
                    malloc(sizeof(struct TSAPdisconnect));
#endif ISO_SUPPORTED
```

```
    conversation->tcp_conv_id = 0;
    conversation->sna_conv_id = 0;
    conversation->network_type = 0;
    conversation->connected = 0;
    conversation->send_mode = NI_NORMAL;
    conversation->recv_mode = NI_NORMAL;
    conversation->priority = NI_NORMAL;

    wait.tv_sec = WAIT_SEC;
    /* configuration parameter defined in localconfig.h      */
    wait.tv_usec = WAIT_U_SEC;
    /* configuration  parameter defined in localconfig.h     */

    ep_info = rsrc_arr[handle];
    /*              end of initialiazation                   */

#ifdef TCP_SUPPORTED
    if (ep_info->tcp_allocated)
    {
        if (conversation->priority == NI_EXPEDITED)
        {
            int toggle = 1;

            if (setsockopt(ep_info->tcp_ep.sock_id,
                SOL_SOCKET, SO_OOBINLINE, (char *) &toggle,
                                        sizeof(int)) == -1)
            {

#ifdef DEBUG
                perror("set socket option");
```

```
#endif DEBUG

                tcp_urg = NI_NORMAL;
            }
            tcp_urg = NI_EXPEDITED;
        }
    }
#endif TCP_SUPPORTED

    if (blocking)
    {


#ifdef DEBUG
        printf("Mode is BLOCKING\n");
#endif DEBUG

        do
        {

#ifdef TCP_SUPPORTED
            if (ep_info->tcp_allocated)
            {

                /*  do a read select to check for any        */
                /*  pending connection                       */
                FD_ZERO(&read_template);
                FD_SET(ep_info->tcp_ep.sock_id,
                                        &read_template);
                status = select(FD_SETSIZE, &read_template,
```

```
    (fd_set *) 0, (fd_set *) 0, &wait);
if (status > 0)
{
    if (FD_ISSET(ep_info->tcp_ep.sock_id,
                            &read_template))
    {
        status =
            accept(ep_info->tcp_ep.sock_id,
                    (struct sockaddr *) 0,
                                (int *) 0);
        if (status > 0)
        {
            int j,
                bytes_2_read;

            /*  fill up the conversation    */
            /*  object for future activities*/
            conversation->tcp_conv_id =
                                    status;
            conversation->network_type =
                                    NI_TCP;
            conversation->connected =
                                    NI_DONE;
            conv_arr[curr_conv_index] =
                                conversation;
            conv_avail[curr_conv_index] = 1;
            *Cv_id = curr_conv_index + 1;
            /* add 1 to conversation index  */
```

```
                    /*  get partner's name          */
                    *partner =
                        get_prtnr_name(*Cv_id,
                                            sz_prtn);

                    if (*sz_prtn > 0)
                    {
                        int status;

                        conversation->connected
                                            = NI_DONE;
                        status =
                            recv_send_priority(*Cv_id,
                                conv_type.expedited);
                        if (status == 1)
                        {
                            conversation->priority =
                                conv_type.expedited;
                            return OK;
                        }
                        else if (status == 0)
                        {
                            conversation->priority =
                                            NI_NORMAL;
                            return OK;
                        }

#ifdef DEBUG
        printf("eaccept : partner priority not received\n");
#endif DEBUG
```

```
                              }

#ifdef DEBUG
        printf("eaccept : partner info not received or\n");
#endif DEBUG

                              conv_avail[curr_conv_index] = 0;
                              free(conversation);
                              return ACCEPT_FAILED;
                          }
                          else
                          {
                              perror("accept");
                              free(conversation);
                              return ACCEPT_FAILED;
                          }
                      }
                  }

              }
#endif TCP_SUPPORTED

#ifdef SNA_SUPPORTED
            if (ep_info->sna_allocated)
            {
                /*              do something                    */
            }
#endif SNA_SUPPORTED
```

```
#ifdef ISO_SUPPORTED
            if (ep_info->osi_allocated)
            {
                char *vec[4];

                /* the implementation does not need more     */
                /* than 4 although never mentioned           */
                /* anywhere. Sort of hack!!                  */

                int vecp,
                    nfds = 0;
                fd_set rfds,
                    wfds,
                    efds;


                if (TNetAccept(&vecp, vec, nfds, NULLFD,
                        NULLFD, NULLFD, OK, td) == NOTOK)
                {

#ifdef DEBUG
        printf("OSI : error in accept, %s %s\n",
                        TErrString(td->td_reason),
                                                td->td_data);
#endif DEBUG

                    free(conversation);
                    return ACCEPT_FAILED;
                }
```

```
                if (vecp > 0)
                {
                    int expedited;

                    if (TInit(vecp, vec, ts, td) == NOTOK)
                    {

#ifdef DEBUG
        printf("OSI :error in init, %s\n",
                                    TErrString(td->td_reason));
#endif DEBUG

                        free(conversation);
                        return ACCEPT_FAILED;
                    }
                    if ((conv_type.expedited == NI_EXPEDITED)
                            && (ts->ts_expedited))
                        expedited = 1;
                    /* both sides want expedited support    */
                    else
                        expedited = 0;
                    if (TConnResponse(ts->ts_sd, NULLTA,
                                        expedited,
                                        ((char *) 0),
                                        0, NULLQOS, td)
                                                    == NOTOK)
                    {

#ifdef DEBUG
```

```
        printf("OSI :error in connection response, %s\n",
                                TErrString(td->td_reason));
#endif DEBUG

                                free(conversation);
                                return ACCEPT_FAILED;
                            }
                            else
                            {
                                int j;

#ifdef DEBUG
        printf("OSI :connection established");
        printf(",getting partner's name\n");
#endif DEBUG

                                conversation->tsap_conv_id =
                                                        ts->ts_sd;
                                conversation->network_type =
                                                        NI_OSI;
                                conversation->connected = NI_DONE;
                                if (expedited)
                                    conversation->priority =
                                                    NI_EXPEDITED;
                                else
                                    conversation->priority =
                                                        NI_NORMAL;
                                conv_arr[curr_conv_index] =
                                                    conversation;
                                conv_avail[curr_conv_index] = 1;
```

```
                        *Cv_id = curr_conv_index + 1;
                        /* add 1 to conv index              */


                        /*      get partner's name          */
                        *partner =
                            get_prtnr_name(*Cv_id, sz_prtn);
                        if (*sz_prtn > 0)
                            return OK;

#ifdef DEBUG
        printf("eaccept OSI: partner name not received \n");
#endif DEBUG

                        conv_avail[curr_conv_index] = 0;
                        free(conversation);
                        return ACCEPT_FAILED;
                    }
                }
            }
#endif ISO_SUPPORTED
        } while (TRUE);
    }           /*      end of  blocking                    */
    else
    {


#ifdef DEBUG
        printf("Mode is NON BLOCKING\n");
#endif DEBUG
```

```
#ifdef TCP_SUPPORTED
        if (ep_info->tcp_allocated)
        {

            /*      do a read select to check for any       */
            /*      pending connection                      */
            FD_ZERO(&read_template);
            FD_SET(ep_info->tcp_ep.sock_id, &read_template);
            status = select(FD_SETSIZE, &read_template,
                (fd_set *) 0, (fd_set *) 0, &wait);
            if (status > 0)
            {
                /*  a pending request for connection        */
                if (FD_ISSET(ep_info->tcp_ep.sock_id,
                                            &read_template))
                {
                    status = accept(ep_info->tcp_ep.sock_id,
                        (struct sockaddr *) 0, (int *) 0);
                    if (status > 0)
                    {
                        int j;

                        conversation->tcp_conv_id = status;
                        conversation->network_type = NI_TCP;
                        conv_arr[curr_conv_index] =
                                            conversation;
                        conversation->connected = NI_DONE;
                        conv_avail[curr_conv_index] = 1;
```

```
            *Cv_id = curr_conv_index + 1;


            /*      get partner's name          */
            *partner =
                get_prtnr_name(*Cv_id, sz_prtn);
            if (*sz_prtn > 0)
            {
                int status;

                conversation->connected = NI_DONE;
                status =
                        recv_send_priority(*Cv_id,
                            conv_type.expedited);
                if (status == 1)
                {
                    conversation->priority =
                        conv_type.expedited;
                    return OK;
                }
                else if (status == 0)
                {
                    conversation->priority =
                                        NI_NORMAL;
                    return OK;
                }

#ifdef DEBUG
    printf("eaccept : partner priority not received\n");
#endif DEBUG
```

```
                        }

#ifdef DEBUG
    printf("eaccept TCP:size of partner name not received\n");
#endif DEBUG

                        conv_avail[curr_conv_index] = 0;
                        free(conversation);
                        return ACCEPT_FAILED;
                    }
                    else
                    {

#ifdef DEBUG
                        perror("TCP accept");
#endif DEBUG

                        free(conversation);
                        return ACCEPT_FAILED;
                    }
                }
                else
                {
                    free(conversation);
                    return REPEAT_LATER;
                }
            }
            else
            {
                free(conversation);
```

```
                    return REPEAT_LATER;
                }

            }
#endif TCP_SUPPORTED


#ifdef SNA_SUPPORTED
            if (ep_info->sna_allocated)
            {
            }
#endif SNA_SUPPORTED



#ifdef ISO_SUPPORTED
            if (ep_info->osi_allocated)
            {
                char *vec[4];

                /*  the implementation does not need more     */
                /*  than 4 although never mentioned           */
                /*  anywhere. Sort of hack!!                  */
                int vecp,
                    nfds = 0;
                fd_set rfds,
                    wfds,
                    efds;

                struct TSAPdisconnect *td;
```

```
            struct TSAPstart *ts;

            td = (struct TSAPdisconnect *)
                malloc(sizeof(struct TSAPdisconnect));
            ts = (struct TSAPstart *)
                malloc(sizeof(struct TSAPstart));

            if (TNetAccept(&vecp, vec, nfds, NULLFD,
                    NULLFD, NULLFD, OK, td) == NOTOK)
            {

#ifdef DEBUG
    printf("OSI : error in accept, %s\n",
                            TErrString(td->td_reason));
#endif DEBUG

                free(conversation);
                return ACCEPT_FAILED;
            }

            if (vecp > 0)
            {
                if (TInit(vecp, vec, ts, td) == NOTOK)
                {

#ifdef DEBUG
    printf("OSI :error in init, %s\n",
                            TErrString(td->td_reason));
#endif DEBUG
```

```
                    free(conversation);
                    return ACCEPT_FAILED;
                }
                else
                {
                    int expedited;

                    if ((conv_type.expedited ==
                                NI_EXPEDITED) &&
                                (ts->ts_expedited))
                        expedited = 1;
                    /* both sides want expedited        */
                    else
                        expedited = 0;
                    if (TConnResponse(ts->ts_sd, NULLTA,
                            expedited, ((char *) 0), 0,
                                    NULLQOS, td) == NOTOK)
                    {

#ifdef DEBUG
    printf("OSI :error in connection response, %s\n",
                            TErrString(td->td_reason));
#endif DEBUG

                        free(conversation);
                        return ACCEPT_FAILED;
                    }
                    else
                    {
                        int j;
```

```
#ifdef DEBUG
    printf("OSI :connection established");
    printf(",getting partner's name\n");
#endif DEBUG

                        conversation->tsap_conv_id =
                                        ts->ts_sd;
                        conversation->network_type =
                                        NI_OSI;
                        conversation->connected =
                                        NI_DONE;
                        if (expedited)
                            conversation->priority =
                                        NI_EXPEDITED;
                        else
                            conversation->priority =
                                        NI_NORMAL;
                        conv_arr[curr_conv_index] =
                                        conversation;
                        conv_avail[curr_conv_index] = 1;
                        *Cv_id = curr_conv_index + 1;
                        /* add 1 to conversation index  */

                        *partner =
                        get_prtnr_name(*Cv_id, sz_prtn);
                        if (*sz_prtn > 0)
                        {
                            return OK;
                        }
```

```
#ifdef DEBUG
    printf("eaccept OSI: partner name not received \n");
#endif DEBUG

                        conv_avail[curr_conv_index] = 0;
                        free(conversation);
                        return ACCEPT_FAILED;
                    }   /* TConnResponse */
                }   /* TInit */
            }   /* vecp > 0 */
            free(conversation);
            return REPEAT_LATER;
        }       /* if ep_info->osi_allocated */
#endif ISO_SUPPORTED
    }           /* else if non blocking */

}
```

```
/************************************************************
*                      econnect.c                           *
*                                                           *
*           Author Debashish Chatterjee                     *
*           Date   19th July 1990                           *
*                                                           *
*   int econnect(my_name, partner_name, blocking,           *
*                                   conv_type, Cv_id)       *
*   "econnect" connects to a remote process in a blocking or *
*   non-blocking mode over a conversation                   *
*   Input:  The symbolic name of this process, a string     *
*           "my_name"                                       *
*           Blocking or non-blocking mode "blocking"        *
*           The symbolic name of the partner "partner_name", *
*           advertised already by the partner               *
*           Kind of conversation by the server and client   *
*           using "conv_type"                               *
*   Output : The conversation identifier "Cv_id" of type    *
*           NA_conv                                         *
*   Returns OK if successful. Otherwise CONNECT_ERROR if    *
*   failed while trying to connect due to invalid partner   *
*   name or non existing partner.                           *
*   If maximum allowable resources have been used up        *
*   NO_RESOURCE is returned.                                *
*   If the mode is non-blocking  and the call could not be  *
*   launched REPEAT_LATER is returned. The caller should try *
*   again.                                                  *
*                                                           *
*************************************************************/
```

```
/*  include files from the include directory                         */
#include "global.h"
#include "shared.h"

extern int rtrv_prtnr_rsrc();
extern int ewrite();
```

```
#ifdef TCP_SUPPORTED
/************************************************************
*                                                           *
*       This function exchanges with the responder the      *
*       priority once the association has been established *
*       This is required only for TCP protocol.             *
*       Sets the priority of the conversation apppropriately*
*       If there is an error, -1 is returned                *
*       If there is a match 1 is returned else 0            *
*                                                           *
************************************************************/
int send_recv_priority(Cv_id, priority)
NA_conv Cv_id;  /* the conversation active                  */
int priority;   /* priority sought by the caller            */
{
    int j,
        prtnr_pr;   /*      partner's priority              */

    j = sizeof(int);
    /* send own priority                                    */
    if (ewrite(Cv_id, BLOCKING, NI_NORMAL,
                                        &priority, &j) == OK)
    {
        if (j == sizeof(int))
        {
            /* receive prtnr's priority                     */
            if (eread(Cv_id, BLOCKING, NI_NORMAL,
                                        &prtnr_pr, &j) == OK)
            {
                if (j == sizeof(int))
```

```
                {

#ifdef DEBUG
    printf("econnect: partner's priority is %d\n", prtnr_pr);
#endif DEBUG

                    if (priority == prtnr_pr)
                        return 1;
                    /*          both agree                      */
                    return 0;
                    /*          mismatch                        */
                }
            }
        }
    }
    return -1;      /*          error in protocol               */
}

#endif TCP_SUPPORTED
```

```
/**************************************************************
*   This procedure sends own name to the remote process       *
**************************************************************/
int send_my_name(Cv_id, my_name)
NA_conv Cv_id;  /*      active conversation identifier      */
char *my_name;  /*      my name to be sent to the partner   */
{

    int max_j,
        j,
        sz_j;


    max_j = strlen(my_name) + 1;
    j = max_j;
    sz_j = sizeof(j);

#ifdef DEBUG
    printf("j,sz_j,max_j: %d, %d, %d \n", j, sz_j, max_j);
#endif DEBUG

    /*      first transmit the size of partner name         */
    if (ewrite(Cv_id, BLOCKING, NI_NORMAL, &j, &sz_j) == OK)
    {
        if (sz_j == sizeof(int))
        /*          successfully transmitted                */
        {
            /*          now the actual name                 */
            if (ewrite(Cv_id, BLOCKING, NI_NORMAL,
                                        my_name, &j) == OK)
```

```
            {
                if (j == max_j)
                    return 0;
            }

#ifdef DEBUG
    printf("econnect/retry : my name could not sent \n");
#endif DEBUG
        }

#ifdef DEBUG
    printf("econnect/retry:name length could not sent\n");
#endif DEBUG
    }
    return -1;
}
```

```
/***********************************************************
*               the econnect function                       *
***********************************************************/
int econnect(my_name, partner_name, blocking,
                                        conv_type, Cv_id)
char *my_name,      /*      name of the local process       */
   *partner_name;   /*      name of the remote process      */
int blocking;       /*      BLOCKING or NON_BLOCKING mode   */
struct conv_char conv_type; /*  conversation characteristics*/
NA_conv *Cv_id;     /*  the conversation identifier returned*/


{
    /* Finds out physical address of the partner to         */
    /* connect from nameserver                              */
    /* From the physical address use appropriate network    */
    /* protocol to connect                                  */

    /* If more than one common network is supported both    */
    /* at local and remote machine , try to connect to      */
    /* any one by                                           */
    /* 1) allocating a communication endpoint               */

    /* 2) If blocking,return if an accept is acknowledged   */
    /* from remote site                                     */
    /* 3) If non-blocking issue a protocol specific call. If*/
    /* the call does not complete and is in progress return */

    /* On a proper connect, returns conversation id  and    */
    /* OK                                                   */
```

```
    fd_set write_template;  /*  write mask for select calls */
    struct timeval wait;    /*  structure for passing time  */
    NA_conn_rsrc *ep_info;  /*  a communication resource    */
    NA_conv_info *conversation; /*  conversation object     */
    int cycle_sel = 0;  /* to select the order in which to  */
                        /* the protocols                    */

#ifdef TCP_SUPPORTED
    struct sockaddr_in conn_to; /*  TCP specific address    */

#endif TCP_SUPPORTED
    int sock,
        status,
        curr_conv_index;
    int which_net = NONE;

#ifdef ISO_SUPPORTED
    /*          OSI specific    declarations                */
    struct TSAPconnect *tc; /*  connection  structure       */
    struct TSAPdisconnect *td;  /*  disconnect structure    */

#endif ISO_SUPPORTED


    /*                  Initialize                          */
#ifdef CTHREADS_SUPPORTED
    if (blocking == BLOCKING)
        mutex_lock(na_conv_lock);
    /*              blocks thread till success              */
```

```
    else
    {
        if (mutex_try_lock(na_conv_lock) == FALSE)
            return REPEAT_LATER;
    }
#endif CTHREADS_SUPPORTED

    for (curr_conv_index = 0;
            curr_conv_index < MAX_CONVERSATIONS;
                                        curr_conv_index++)
        if (!conv_avail[curr_conv_index])
            break;

#ifdef CTHREADS_SUPPORTED
    mutex_unlock(na_conv_lock);
#endif CTHREADS_SUPPORTED

    if (curr_conv_index == MAX_CONVERSATIONS)
        return NO_RESOURCE;
    conversation = (NA_conv_info *)
                    malloc((unsigned) sizeof(NA_conv_info));

    conversation->tsap_conv_id = 0;
    conversation->tcp_conv_id = 0;
    conversation->sna_conv_id = 0;
    conversation->network_type = 0;
    conversation->connected = 0;
    conversation->send_mode = NI_NORMAL;
    conversation->recv_mode = NI_NORMAL;
    conversation->my_name = malloc(strlen(my_name) + 1);
```

```
    strcpy(conversation->my_name, my_name);

    wait.tv_sec = WAIT_SEC;
    /* configuration parameter defined in localconfig.h     */
    wait.tv_usec = WAIT_U_SEC;
    /* configuration parameter defined in localconfig.h     */

    ep_info = (NA_conn_rsrc *)
                malloc((unsigned) sizeof(NA_conn_rsrc));

#ifdef ISO_SUPPORTED
    isodetailor(my_name, 1);
    td = (struct TSAPdisconnect *)
        malloc(sizeof(struct TSAPdisconnect));
    tc = (struct TSAPconnect *)
        malloc(sizeof(struct TSAPconnect));
#endif ISO_SUPPORTED

    /* from the network directory ,retrieve using            */
    /* partner_name the real network & the physical          */
    /* address ie partner_name  ---> NA_conn_rsrc of         */
    /* partner ; then proceed                                */

    if (rtrv_prtnr_rsrc(partner_name, ep_info) < 0)
    {

#ifdef DEBUG
        printf("Partner look up in nameserver failed \n");
#endif DEBUG
```

```
        free(ep_info);
        free(conversation);
        return NO_SUCH_PARTNER;
    }

#ifdef TCP_SUPPORTED
    if (TCP_available && ep_info->tcp_allocated)
    {
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0)
        {

#ifdef DEBUG
            printf("TCP socket failed\n");
#endif DEBUG
        }
        else
        {
            int toggle = 1;

            /*  set socket option to receive urgent data     */
            if (setsockopt(sock, SOL_SOCKET, SO_OOBINLINE,
                    (char *) &toggle, sizeof(int)) == 0)
            {

                conn_to.sin_family = AF_INET;
                bcopy((char *)
                        &(ep_info->tcp_ep.in_socket.sin_addr),
                        (char *) &conn_to.sin_addr,
                        sizeof(ep_info->tcp_ep.
```

```
                                        in_socket.sin_addr));
                conn_to.sin_port =
                        ep_info->tcp_ep.in_socket.sin_port;

                /* asynchronous call to connect                 */
                if (fcntl(sock, F_SETFL, FNDELAY) < 0)
                {

#ifdef DEBUG
                    perror("fcntl F_SETFL,FNDELAY");
#endif DEBUG
                }
                else
                    which_net += NI_TCP;
            }   /*            setsockopt                        */
            else
            {

#ifdef DEBUG
                perror("set socket option");
#endif DEBUG
            }
        }
    }
#endif TCP_SUPPORTED

#ifdef SNA_SUPPORTED
    if (SNA_available && ep_info->sna_allocated)
    {
        which_net += NI_SNA;
```

```
    }
#endif SNA_SUPPORTED

#ifdef ISO_SUPPORTED
    if (OSI_available && ep_info->osi_allocated)
    {
        which_net += NI_OSI;
    }
#endif ISO_SUPPORTED

#ifndef DEBUG1

#ifdef TCP_SUPPORTED && ISO_SUPPORTED
    if (which_net == OSI_TCP || which_net == ALL)
        cycle_sel = random() & 01;
    /*  if cycle_sel is 0 then the order is TCP-->OSI-->SNA */
    /*  if cycle_sel is 1 then the order is OSI-->TCP-->SNA */
#endif TCP_SUPPORTED && ISO_SUPPORTED

#endif DEBUG1

    if (blocking)
    {

#ifdef TCP_SUPPORTED

#ifdef ISO_SUPPORTED
        if (cycle_sel)
            goto L1;
#endif ISO_SUPPORTED
```

```
#endif TCP_SUPPORTED

        do
        {

#ifdef TCP_SUPPORTED

#ifdef ISO_SUPPORTED
    L2:
#endif TCP_SUPPORTED

#endif TCP_SUPPORTED

#ifdef TCP_SUPPORTED
            if (which_net == NI_TCP ||
                        which_net == OSI_TCP ||
                            which_net == SNA_TCP ||
                                        which_net == ALL)
            {

#ifdef DEBUG
                printf("TCP blocking\n");
#endif DEBUG

                /*      protocol specific connect call      */
                status =
                    connect(sock, &conn_to, sizeof(conn_to));
                if (status == 0)
                {
```

```
int j,
    max_j,
    sz_j;
char temp[2];

/* since a connection has been done     */
/* initialise the conversation object   */
conversation->tcp_conv_id = sock;
conversation->network_type = NI_TCP;
conversation->connected = NI_DONE;
free(ep_info);
conv_arr[curr_conv_index] = conversation;
conv_avail[curr_conv_index] = 1;
*Cv_id = curr_conv_index + 1;

/*          transmit own name             */
if (send_my_name(*Cv_id,
        conversation->my_name) == 0)
{
    int status;

    conversation->connected = NI_DONE;
    /*exchange priority to be supported */
    status = send_recv_priority(*Cv_id,
        conv_type.expedited);
    if (status == 1)
    {
        conversation->priority =
                    conv_type.expedited;
        return OK;
```

```
        }
        else if (status == 0)
        {
            conversation->priority =
                                    NI_NORMAL;
            return OK;
        }
    }
    conv_avail[curr_conv_index] = 0;
    free(conversation);
    return CONNECT_FAILED;
}   /* status == 0 */
switch (errno)
{
case EINPROGRESS:
    do
    {
        /*  connection is still in progress */
        FD_ZERO(&write_template);
        FD_SET(sock, &write_template);
        /* select to find if complete       */
        status = select(FD_SETSIZE,
            (fd_set *) 0, &write_template,
            (fd_set *) 0, &wait);
        if (status > 0)
        {
            if (FD_ISSET(sock,
                            &write_template))
            {
                if (sock == -1)
```

```
{
    continue;
}
conversation->tcp_conv_id =
                        sock;
conversation->network_type =
                    NI_TCP;
conversation->connected =
                    NI_DONE;
free(ep_info);
conv_arr[curr_conv_index] =
                conversation;
conv_avail[curr_conv_index] =
                            1;
*Cv_id = curr_conv_index + 1;

/*  transmit own name       */
if (send_my_name(*Cv_id,
        conversation->my_name)
                        == 0)
{
    int status;

    conversation->connected =
                    NI_DONE;
    status =
/*  exchange priority       */
    send_recv_priority(*Cv_id,
        conv_type.expedited);
    if (status == 1)
```

```
                                        {
                                            conversation->priority
                                            = conv_type.expedited;
                                            return OK;
                                        }
                                        else if (status == 0)
                                        {
                                            conversation->priority
                                                    = NI_NORMAL;
                                            return OK;
                                        }
                                    }
                                    conv_avail[curr_conv_index] =
                                                                0;
                                    free(conversation);
                                    return CONNECT_FAILED;
                                }
                            }
                        } while (TRUE);
                    case EALREADY:
                        break;
                    default:
                        {

#ifdef DEBUG
                            perror("connect:");
#endif DEBUG

                            free(conversation);
                            return CONNECT_FAILED;
```

```
                    }
                }
            }
#endif TCP_SUPPORTED

#ifdef SNA_SUPPORTED
            if (which_net == NI_SNA ||
                    which_net == OSI_SNA ||
                        which_net == SNA_TCP ||
                                which_net == ALL)
            {
            /*          do something                                */
            }
#endif SNA_SUPPORTED

#ifdef TCP_SUPPORTED

#ifdef ISO_SUPPORTED
    L1:
#endif ISO_SUPPORTED

#endif TCP_SUPPORTED

#ifdef ISO_SUPPORTED
            if (which_net == NI_OSI ||
                    which_net == OSI_TCP ||
                        which_net == OSI_SNA ||
                                which_net == ALL)
            {
                int expedited;
```

```
#ifdef DEBUG
                printf("OSI blocking\n");
#endif DEBUG

                if (conv_type.expedited == NI_EXPEDITED)
                    expedited = 1;
                else
                    expedited = 0;
                /* protocol specific non-blocking connect    */
                if ((status =
                        TAsynConnRequest(NULLTA,
                                        ep_info->tsap_ep,
                                        expedited,
                                        ((char *) 0), 0,
                                        NULLQOS, tc, td, 1))
                                                    == NOTOK)
                {

#ifdef DEBUG
    printf("OSI error in Connection Request, %s\n",
                            TErrString(td->td_reason));
#endif DEBUG

                    free(ep_info);
                    free(conversation);
                    return CONNECT_FAILED;
                }   /*      TConnRequest == NOTOK            */
                else
                {
```

```
                        /*  connection request in progress              */
                            int i,
                                j,
                                max_j,
                                sz_j;
                            int trans_d;

                            trans_d = tc->tc_sd;
                            while (status == CONNECTING_1 ||
                                            status == CONNECTING_2)
                            {
                                int nfds;
                                fd_set mask,
                                   *rmask,
                                   *wmask;

                                nfds = 0;
                                FD_ZERO(&mask);
                                if ((status =
                                        TSelectMask(trans_d, &mask,
                                                &nfds, td)) == NOTOK)
                                {

#ifdef DEBUG
    printf("error in TSelect 1, %s, %s\n",
                    TErrString(td->td_reason),
                                td->td_data);
#endif DEBUG

                                    free(ep_info);
```

```
            free(conversation);
            return CONNECT_FAILED;
        }

        /*  set the read and write masks    */
        rmask = (status == CONNECTING_2) ?
                            &mask : NULLFD;
        wmask = (status == CONNECTING_2)
                            ? NULLFD : &mask;

        if ((xselect(nfds, rmask, wmask,
                        NULLFD, 1)) == NOTOK)
        {

#ifdef DEBUG
    printf("error in xselect 1, %s, %s\n",
                    TErrString(td->td_reason),
                                td->td_data);
#endif DEBUG

            free(ep_info);
            free(conversation);
            return CONNECT_FAILED;
        }

        if ((rmask &&
                FD_ISSET(trans_d, rmask) == 0)
            || (wmask &&
                FD_ISSET(trans_d, wmask) == 0))
            continue;
```

```
                    /*  still in progress; not ready    */
                    /*  for read and write activities   */

                    if ((status =
                            TAsynRetryRequest(trans_d,
                                                    tc, td))
                                                    == NOTOK)
                    {

#ifdef DEBUG
    printf("error in TAsynRetryRequest, %s, %s\n",
                            TErrString(td->td_reason),
                                                td->td_data);
#endif DEBUG

                        free(ep_info);
                        free(conversation);
                        return CONNECT_FAILED;
                    }


                }   /*              while                   */
                /*          conversation established        */
                /*  fill conversation object for future */
                /*  references                              */
                conversation->tsap_conv_id = trans_d;
                conversation->network_type = NI_OSI;
                conversation->connected = NI_DONE;
                if (tc->tc_expedited)
                    conversation->priority = NI_EXPEDITED;
```

```
                else
                    conversation->priority = NI_NORMAL;
                free(ep_info);
                conv_arr[curr_conv_index] = conversation;
                conv_avail[curr_conv_index] = 1;
                *Cv_id = curr_conv_index + 1;

                if (send_my_name(*Cv_id,
                            conversation->my_name) == 0)
                    return OK;
            }   /*      ConnRequest == OK                */
            conv_avail[curr_conv_index] = 0;
            free(conversation);
            return CONNECT_FAILED;
        }   /*              which net                    */
#endif ISO_SUPPORTED

        } while (TRUE);
    }
    else
    {           /*      start non-blocking code          */

#ifdef TCP_SUPPORTED

#ifdef ISO_SUPPORTED
        if (cycle_sel)
            goto L1a;
L2a:
#endif ISO_SUPPORTED
```

```
#endif TCP_SUPPORTED

#ifdef TCP_SUPPORTED
        if (which_net == NI_TCP ||
                    which_net == OSI_TCP ||
                            which_net == SNA_TCP ||
                                        which_net == ALL)
        {

#ifdef DEBUG
            printf("TCP nonblocking\n");
#endif DEBUG

            /*  TCP specific connect call                      */
            status = connect(sock, &conn_to, sizeof(conn_to));
            if (status == 0)
            {
                int j;
                int max_j,
                    sz_j;

                /*      conversation established               */
                conversation->tcp_conv_id = sock;
                conversation->network_type = NI_TCP;
                conversation->connected = NI_DONE;
                free(ep_info);
                conv_arr[curr_conv_index] = conversation;
                conv_avail[curr_conv_index] = 1;
                *Cv_id = curr_conv_index + 1;
```

```
        if (send_my_name(*Cv_id,
                         conversation->my_name) == 0)
        {
            int status;

            conversation->connected = NI_DONE;
            status = send_recv_priority(*Cv_id,
                                        conv_type.expedited);
            if (status == 1)
            {
                conversation->priority =
                                        conv_type.expedited;
                return OK;
            }
            else if (status == 0)
            {
                conversation->priority = NI_NORMAL;
                return OK;
            }
        }
        conv_avail[curr_conv_index] = 0;
        free(conversation);
        return CONNECT_FAILED;
    }
    switch (errno)
    {
    case EWOULDBLOCK:
        {
            free(ep_info);
            free(conversation);
```

```
                    return REPEAT_LATER;
                }
            case EINPROGRESS:
                conversation->tcp_conv_id = sock;
                conversation->network_type = NI_TCP;
                free(ep_info);
                conv_arr[curr_conv_index] = conversation;
                conv_avail[curr_conv_index] = 1;
                *Cv_id = curr_conv_index + 1;
                conversation->connected = NI_INPROGRESS;
                conversation->priority =
                                    conv_type.expedited;
                return IN_PROGRESS;
            case EALREADY:
                free(ep_info);
                free(conversation);
                return REPEAT_LATER;
            default:

#ifdef DEBUG
                perror("connect:");
#endif DEBUG

                free(ep_info);
                free(conversation);
                return CONNECT_FAILED;
            }
        }
#endif TCP_SUPPORTED
```

```
#ifdef SNA_SUPPORTED
        if (which_net == NI_SNA ||
                    which_net == OSI_SNA ||
                            which_net == SNA_TCP ||
                                        which_net == ALL)
        {
        }
#endif SNA_SUPPORTED

#ifdef TCP_SUPPORTED

#ifdef ISO_SUPPORTED
        if (cycle_sel)
            goto L3a;

L1a:
#endif ISO_SUPPORTED

#endif TCP_SUPPORTED

#ifdef ISO_SUPPORTED
        if (which_net == NI_OSI ||
                    which_net == OSI_TCP ||
                            which_net == OSI_SNA ||
                                        which_net == ALL)
        {
            struct TSAPconnect *tc;
            struct TSAPdisconnect *td;
            int expedited;
```

```
#ifdef DEBUG
            printf("OSI nonblocking\n");
#endif DEBUG

            td = (struct TSAPdisconnect *)
                malloc(sizeof(struct TSAPdisconnect));
            tc = (struct TSAPconnect *)
                malloc(sizeof(struct TSAPconnect));

            if (conv_type.expedited == NI_EXPEDITED)
                expedited = 1;
            else
                expedited = 0;
            /*  osi specific non-blocking call              */
            if ((status =
                    TAsynConnRequest(NULLTA, ep_info->tsap_ep,
                                    expedited, ((char *) 0),
                                    0, NULLQOS, tc, td, 1))
                                                    == NOTOK)
            {

#ifdef DEBUG
    printf("OSI error in Connection Request, %s, %s\n",
                                TErrString(td->td_reason),
                                                td->td_data);
#endif DEBUG

                free(ep_info);
                free(conversation);
                return CONNECT_FAILED;
```

```
}   /* TConnRequest == NOTOK                          */
else if (status == DONE)
{
    int j,
        max_j,
        sz_j;

    /*      conversation established                  */
    conversation->tsap_conv_id = tc->tc_sd;
    conversation->network_type = NI_OSI;
    conversation->connected = NI_DONE;
    free(ep_info);
    conv_arr[curr_conv_index] = conversation;
    conv_avail[curr_conv_index] = 1;
    *Cv_id = curr_conv_index + 1;

    if (send_my_name(*Cv_id,
                conversation->my_name) == 0)
    {
        if (tc->tc_expedited == 1)
            conversation->priority =
                                NI_EXPEDITED;
        else
            conversation->priority = NI_NORMAL;
        free(tc);
        free(td);
        return OK;
    }
    conv_avail[curr_conv_index] = 0;
    free(conversation);
```

```
                return CONNECT_FAILED;
            }
            else
            {
                conversation->tsap_conv_id = tc->tc_sd;
                conversation->network_type = NI_OSI;
                conversation->connected = status;
                conversation->priority = conv_type.expedited;
                free(ep_info);
                conv_arr[curr_conv_index] = conversation;
                conv_avail[curr_conv_index] = 1;
                *Cv_id = curr_conv_index + 1;
                free(tc);
                free(td);
                return IN_PROGRESS;
            }
        }
#endif ISO_SUPPORTED

#ifdef TCP_SUPPORTED

#ifdef ISO_SUPPORTED
        if (cycle_sel)
            goto L2a;
L3a:
#endif ISO_SUPPORTED

#endif TCP_SUPPORTED

        return REPEAT_LATER;
```

```
    }
}
```

```
/************************************************************
*                        retry.c                            *
*                                                           *
*            Author Debashish Chatterjee                    *
*            Date   19th July 1990                          *
*                                                           *
*    int retry_connect(Cv_id)                               *
*                                                           *
*    retries a nonblocking econnect that has returned       *
*    IN_PROGRESS                                            *
*                                                           *
*                                                           *
*    Input: Cv_id the conversation identifier returned from *
*           non_blocking econnect call.                     *
*    returns OK if connected else CONNECT_ERROR if failed   *
*    while trying to connect due to invalid partner name or *
*    non existing partner.                                  *
*    IN_PROGRESS if connection establishment is still in    *
*    progress.                                              *
*                                                           *
************************************************************/

/* include files from the include directory                */
#include "global.h"
#include "shared.h"

extern int rtrv_prtnr_rsrc();    /*  defined in configure.c  */
extern int eread();              /*  defined in eread.c      */
extern int ewrite();             /*  defined in ewrite.c     */
extern int send_my_name();       /*  defined in econnect.c   */
```

```
extern int send_recv_priority();/*  defined in econnect.c   */
```

```
/**********************************************************
*                  retry_connect function                 *
**********************************************************/
int retry_connect(Cv_id)
NA_conv Cv_id;
/*  conversation identifier of the conversation in progress */

{
    fd_set write_template;
    struct timeval wait;
    NA_conv_info *conversation;
#ifdef DEBUG
    printf("module retry\n");
#endif DEBUG
    if ((Cv_id < 1) || (Cv_id > MAX_CONVERSATIONS))
        return BAD_C_HANDLE;    /*  messed up or masked     */
    Cv_id--;
    if (conv_avail[Cv_id] == 0)
        return BAD_C_HANDLE;
    /*  handle does not reference any active conversation   */

    conversation = conv_arr[Cv_id];

    if (conversation->connected == NI_DONE)
        return OK;

#ifdef TCP_SUPPORTED
    if (conversation->connected != NI_INPROGRESS)
        return NI_BAD_OPERATION;
```

```
#endif TCP_SUPPORTED


#ifdef TCP_SUPPORTED
    wait.tv_sec = WAIT_SEC;
    wait.tv_usec = WAIT_U_SEC;
    /* configuration parameter defined in localconfig.h     */

    if (conversation->network_type == NI_TCP)
    {
        int status;
        FD_ZERO(&write_template);
        FD_SET(conversation->tcp_conv_id, &write_template);
        /*  select to find if conversation is connected     */
        status = select(FD_SETSIZE, (fd_set *) 0,
                        &write_template,
                        (fd_set *) 0, &wait);
        if (status > 0)
        {
            if (FD_ISSET
                (conversation->tcp_conv_id, &write_template))
            {
                int j;
                int max_j,
                    sz_j;


                /*  transfer own name to partner             */
                conversation->connected = NI_DONE;
                if (send_my_name(Cv_id + 1,
```

```
                        conversation->my_name) == 0)
                {
                    int status;
                    conversation->connected = NI_DONE;
                    /*  exchange priority                       */
                    status = send_recv_priority(Cv_id + 1,
                        conversation->priority);
                    if (status == 1)
                    {
                        return OK;
                    }
                    else if (status == 0)
                    {
                        conversation->priority = NI_NORMAL;
                        return OK;
                    }
                }
            }   /*                    if FD_ISSET               */
            conv_avail[Cv_id] = 0;
            free(conversation);
            return CONNECT_FAILED;
        }       /*                if status                     */
        return IN_PROGRESS;
    }
#endif TCP_SUPPORTED
#ifdef ISO_SUPPORTED
    if (conversation->network_type == NI_OSI)
    {
        struct TSAPconnect *tc;
        struct TSAPdisconnect *td;
```

```
        int nfds,
            status;
        fd_set mask,
           *rmask,
           *wmask;
#ifdef DEBUG
        printf("OSI retry\n");
#endif DEBUG
        td = (struct TSAPdisconnect *)
            malloc(sizeof(struct TSAPdisconnect));
        tc = (struct TSAPconnect *)
            malloc(sizeof(struct TSAPconnect));

        status = conversation->connected;
        if (status == CONNECTING_1 ||
                                    status == CONNECTING_2)
        {
            nfds = 0;
            FD_ZERO(&mask);
            /*  set mask for conversation                          */
            if ((status =
                    TSelectMask(conversation->tsap_conv_id,
                        &mask, &nfds, td)) == NOTOK)
            {
#ifdef DEBUG
                printf("error in TSelect, %s, %s\n",
                                    TErrString(td->td_reason),
                                                td->td_data);
#endif DEBUG
```

```
                free(conversation);
                return CONNECT_FAILED;
            }
            /*      set read and write mask                     */
            rmask =
                (status == CONNECTING_2) ? &mask : NULLFD;
            wmask =
                (status == CONNECTING_2) ? NULLFD : &mask;

            /*      select to see if read/write permissible */
            if ((xselect(nfds, rmask, wmask, NULLFD, 1))
                == NOTOK)
            {
#ifdef DEBUG
                printf("error in xselect, %s, %s\n",
                    TErrString(td->td_reason),
                    td->td_data);
#endif DEBUG
                free(conversation);
                return CONNECT_FAILED;
            }

            if ((rmask && FD_ISSET(conversation->tsap_conv_id,
                        rmask) == 0) ||
                (wmask && FD_ISSET(conversation->tsap_conv_id,
                        wmask) == 0))
                return IN_PROGRESS;

            /*  check to see if conection completed         */
            if ((status =
```

```
            TAsynRetryRequest(conversation->tsap_conv_id,
                    tc, td)) == NOTOK)
        {
#ifdef DEBUG
            printf("error in TAsynRetryRequest, %s, %s\n",
                TErrString(td->td_reason),
                td->td_data);
#endif DEBUG
            free(conversation);
            return CONNECT_FAILED;
        }
        if ((status == CONNECTING_1) ||
                                (status == CONNECTING_2))
        {
            conversation->connected = status;
            return IN_PROGRESS;
        }
        conversation->connected = NI_DONE;
        if (tc->tc_expedited)
            conversation->priority = NI_EXPEDITED;
        else
            conversation->priority = NI_NORMAL;
        if (send_my_name(Cv_id + 1,
                            conversation->my_name) == 0)
            return OK;
        conv_avail[Cv_id] = 0;
        free(conversation);
        return CONNECT_FAILED;
    }
    else
```

```
            return NI_BAD_OPERATION;

    }
#endif ISO_SUPPORTED
}
```

```
/***************************************************************
*                                                              *
*                      terminate.c                             *
*                                                              *
*           Author Debashish Chatterjee                        *
*           Date   19th July 1990                              *
*                                                              *
*       Describes all the association release calls            *
*                                                              *
***************************************************************/


/*          define the include files                          */
#include  "global.h"
#include "shared.h"
```

```
/***********************************************************
*                                                         *
*   function "eclose" closes a conversation normally and *
*   frees up associated memory                            *
*                                                         *
*   Input : a conversation  identifier                    *
*   Output: OK if conversation identifier was correct other- *
*           wise BAD_C_HANDLE                             *
*                                                         *
***********************************************************/
int eclose(Cv_id)
NA_conv Cv_id;  /*  a conversation identifier               */
{
    NA_conv_info *conversation;


    if ((Cv_id < 1) || (Cv_id > MAX_CONVERSATIONS))
        return BAD_C_HANDLE;
    Cv_id--;
    if (conv_avail[Cv_id] == 0)
        return BAD_C_HANDLE;

    conversation = conv_arr[Cv_id];
#ifdef TCP_SUPPORTED
    if (conversation->network_type == NI_TCP)
    {
        close(conversation->tcp_conv_id);
        conv_avail[Cv_id] = 0;
        free(conv_arr[Cv_id]);
```

```
        return OK;
    }
#endif TCP_SUPPORTED
#ifdef SNA_SUPPORTED
    if (conversation->network_type == NI_SNA)
    {
        conv_avail[Cv_id] = 0;
        free(conv_arr[Cv_id]);
        free(conversation);
        return OK;
    }
#endif SNA_SUPPORTED
#ifdef ISO_SUPPORTED
    if (conversation->network_type == NI_OSI)
    {
        struct TSAPdisconnect *td;
        td = (struct TSAPdisconnect *)
            malloc(sizeof(struct TSAPdisconnect));
        if (TDiscRequest
                  (conversation->tsap_conv_id,
                              NULLCP, 0, td) == NOTOK)
        {
        /*      ignore                                      */
#ifdef DEBUG
        printf("OSI :error in disconnect req, %s %s\n",
                        TErrString(td->td_reason),
                                            td->td_data);
#endif DEBUG
        }
        conv_avail[Cv_id] = 0;
```

```
        free(conv_arr[Cv_id]);
        free(conversation);
        free(td);
        return OK;
    }
#endif ISO_SUPPORTED
}
```

```
/************************************************************
*                                                           *
*   function "delete_rsrc" deletes a name mapping and       *
*   frees up associated memory                              *
*                                                           *
*   Input : a handle; communication resource                *
*   Output: OK if communication resource was correct other- *
*           wise BAD_R_HANDLE                               *
*                                                           *
************************************************************/
int delete_rsrc(handle)
NA_rsrc handle;     /*  a handle; communication resource    */
{
#ifdef ISO_SUPPORTED
    struct TSAPdisconnect *td;
#endif ISO_SUPPORTED
    if ((handle < 0) || (handle > MAX_CONVERSATIONS))
        return BAD_R_HANDLE;
    handle--;
#ifdef ISO_SUPPORTED
    if (rsrc_arr[handle]->osi_allocated)
    {
        td = (struct TSAPdisconnect *)
            malloc(sizeof(struct TSAPdisconnect));
        if (TNetClose(rsrc_arr[handle]->tsap_ep,
                                                td) == NOTOK)
        {
#ifdef DEBUG
            printf("OSI : error in TNetClose, %s\n",
```

```
                                        TErrString(td->td_reason));
#endif DEBUG
        }
        free(td);
    }
#endif ISO_SUPPORTED
    free(rsrc_arr[handle]);
    ep_avail[handle] = 0;

    return OK;
}
```

## E.3 Data transfer and multiplexing

```
/**********************************************************
*                         eread.c                         *
*                                                         *
*          Author Debashish Chatterjee                    *
*          Date   19th July 1990                          *
*                                                         *
*   int eread(Cv_id, blocking, priority, data , data_len) *
*   "eread" reads data in a blocking or non-blocking mode *
*   over a conversation                                   *
*   Input : The conversation identifier "Cv_id" of type   *
*           NA_conv  referencing a conversation that has  *
*           already been established                      *
*           Blocking or non-blocking mode "blocking"      *
*           Pointer to a buffer "data" where the data is to *
*           read                                          *
*           "priority" is kind of data expected to be read. *
*           The priority is usually NI_NORMAL. If willingness*
*           to receive expedited data has been hinted and the*
*           previous read has returned status NI_URG_PENDING,*
*           then the priority HAS to be NI_EXPEDITED.     *
*           NI_EXPEDITED can ONLY be used with blocking mode.*
*   Input/output : "data_len" contains the size of the buffer*
*           "data" on input, and actual no of bytes read is *
*           returned on output.                           *
*   BLOCKING Semantics:                                   *
*   The call returns only if the requested number of bytes *
*   "data_len" has been read or an error occurs or urgent *
*   data was received.                                    *
*   If it returns OK then it was successful in reading all *
```

```
*   the data requested.                                       *
*   If "data_len" is less than specified at input and NI_EOT *
*   was returned, the conversation has been closed. The bytes*
*   read prior to termination is returned as "data_len".      *
*   If there was internal problem, READ_ERROR is returned.    *
*   NI_URG_PENDING indicates urgent data has been received,   *
*   and the next read should be a blocking call to read       *
*   urgent data with priority set to NI_EXPEDITED.            *
*   If priority was NI_EXPEDITED, then the "data_len" on      *
*   return reflects the bytes of urgent data read. On no      *
*   account will it be more than MAX_URG_SZ.                  *
*   If it returns OK, no more urgent data is pending. If      *
*   NI_URG_PENDING is returned once again, some more urgent   *
*   data has arrived or is pending.                           *
*   NON-BLOCKING Semantics:                                   *
*   If the mode is non-blocking and the call returns OK then *
*   some data was read and the actual size is specified by    *
*   "data_len".                                               *
*   If NI_EOT was returned, then end of data transmission;    *
*   conversation has been closed by partner.                  *
*   If the no data was available immediately, REPEAT_LATER    *
*   is returned.                                              *
*   The caller should try again.                              *
*   NI_URG_PENDING indicates urgent data has been received,   *
*   and the next read should be a blocking call to read       *
*   urgent data with priority set to NI_EXPEDITED.            *
*                                                             *
*   For both the modes, NI_BAD_PRIORITY indicates wrong use   *
*   of priority, NI_BAD_OPERATION indicates conversation has *
*   not been established as yet, BAD_C_HANDLE indicates a     *
```

```
*   messed up  or masked  conversation identifier               *
*                                                               *
***************************************************************/

/*  include files from the include directory                   */
#include "global.h"
#include "shared.h"
/***************************************************************
*  translates a packet queue to a data stream for OSI reads  *
***************************************************************/

#ifdef ISO_SUPPORTED
rd_buf_q(recv_buf, qb)

char *recv_buf;      /* a stream of bytes when it returns     */
struct qbuf *qb;     /* data received as packets and queued  */
                     /* in that order                         */

{
    int i = 0;

    while (qb->qb_len > 0)
    {
        bcopy(qb->qb_data, &(recv_buf[i]), qb->qb_len);
        i += qb->qb_len;
        qb = qb->qb_forw;
    }

}
```

```
#endif ISO_SUPPORTED
```

```
/**********************************************************
*                 the eread function                      *
**********************************************************/
int eread(Cv_id, blocking, priority, data, data_len)

NA_conv Cv_id;  /* Conversation identifier                  */
int blocking;   /* Blocking or Non-blocking read operation  */
int priority;   /* NI_NORMAL or NI_EXPEDITED                */
char *data;     /* Pointer to buffer where the data is read */
int *data_len;  /* Size of data  to be read; also the size  */
                /* of the buffer "data"                     */

{
   NA_conv_info *conversation;
   /* the conversation structure referring to Cv_id         */

#ifdef TCP_SUPPORTED
   int tcp_status; /* status returned by TCP calls          */
   int expect_urg, /* when true checks for urgent data      */
       return_urg; /* when true returns urgent data         */
   fd_set read_template,   /* socket read mask for          */
                           /*  multiplexing                 */
       urg_template;    /* socket exceptional maskfor urgent*/
                        /* data                             */
   struct timeval wait;    /* UNIX time structure           */

#endif TCP_SUPPORTED

#ifdef ISO_SUPPORTED
```

```
    struct TSAPdisconnect *td;
    struct TSAPdata *tx_recv;

#endif ISO_SUPPORTED

    if ((priority == NI_EXPEDITED) &&
                        (blocking == NON_BLOCKING))
        return NI_BAD_PRIORITY;
    if ((Cv_id < 1) || (Cv_id > MAX_CONVERSATIONS))
        return BAD_C_HANDLE;
    Cv_id--;
    if (conv_avail[Cv_id] == 0)
        return BAD_C_HANDLE;

    conversation = conv_arr[Cv_id];
    if (conversation->connected != NI_DONE)
        return NI_BAD_OPERATION;    /* conversation not      */
                                    /* established yet       */


    switch (conversation->priority)
    {
    case NI_EXPEDITED:
        if (priority == NI_NORMAL)
        {
            if (conversation->recv_mode == NI_EXPEDITED)
                return NI_BAD_PRIORITY;
            /* with urgent data pending normal read  not    */
            /* permitted                                     */
```

```
#ifdef TCP_SUPPORTED
            expect_urg = TRUE;
            return_urg = FALSE;
#endif TCP_SUPPORTED
        }
        else if (priority == NI_EXPEDITED)
        {

#ifdef TCP_SUPPORTED
            expect_urg = TRUE;
            return_urg = TRUE;
#endif TCP_SUPPORTED
        }
        else
        {
            return NI_BAD_PRIORITY;
        }
        break;
    case NI_NORMAL:
        if (priority != NI_NORMAL)
        {
            return NI_BAD_PRIORITY;
        }
        else
        {

#ifdef TCP_SUPPORTED
            expect_urg = FALSE;
            return_urg = FALSE;
#endif TCP_SUPPORTED
```

```
        }
    }

#ifdef TCP_SUPPORTED
    wait.tv_sec = WAIT_SEC;
    /* configuration parameter defined in localconfig.h      */
    wait.tv_usec = WAIT_U_SEC;
    /* configuration parameter defined in localconfig.h      */
#endif TCP_SUPPORTED

#ifdef ISO_SUPPORTED
    td = (struct TSAPdisconnect *)
                malloc(sizeof(struct TSAPdisconnect));
    tx_recv = (struct TSAPdata *)
                malloc(sizeof(struct TSAPdata));
#endif ISO_SUPPORTED

    if (blocking)
    {

#ifdef TCP_SUPPORTED
        if (conversation->network_type == NI_TCP)
        {
            int d_buf_ptr = 0;
            /* pointer to the position in the data          */
            /* buffer for subsequent reads                  */
            int sz_2_b_read;
            /* size of data to be read for that call        */

            /* initialise the data len to be read           */
```

```
        sz_2_b_read = *data_len;
        *data_len = 0;
        do
        {
            FD_ZERO(&read_template);
            FD_ZERO(&urg_template);
            FD_SET(conversation->tcp_conv_id,
                                            &read_template);
            if (expect_urg == TRUE)
                FD_SET(conversation->tcp_conv_id,
                                            &urg_template);

            /*  check to see if data pending             */
            tcp_status = select(FD_SETSIZE,
                                &read_template,
                                (fd_set *) 0,
                                &urg_template,
                                &wait);

            if (tcp_status > 0)
            {
                int urg_mark = 0;

                /*      is this the urgent byte          */
                if (ioctl(conversation->tcp_conv_id,
                        SIOCATMARK,
                        (char *) &urg_mark) < 0)
                {

#ifdef DEBUG
```

```
                        perror("ioctl:");
#endif DEBUG

                        return READ_ERROR;
                    }

                    if ((urg_mark == 1) &&
                        (FD_ISSET(conversation->tcp_conv_id,
                                  &urg_template)))
                    {
                        char c;

                        if (return_urg == FALSE)
                            return NI_URG_PENDING;
                        if (conversation->recv_mode ==
                                                  NI_EXPEDITED)
                /* the delimiter marking end of urgent data */
                        {
                            tcp_status =
                                    read(conversation->
                                                  tcp_conv_id,
                                                      &c, 1);
                            if (tcp_status != 1)
                            {

#ifdef DEBUG
                    perror("urg marker ends read error");
#endif DEBUG

                                return READ_ERROR;
```

```
                    }
                    conversation->recv_mode =
                                            NI_NORMAL;
                    return OK;
                }
                else
                {
                    /* first urgent read request    */
                    /* after NI_URG_PENDING was     */
                    /* returned                     */
                    tcp_status =
                            read(conversation->
                                        tcp_conv_id,
                                            &c, 1);
                    if (tcp_status != 1)
                    {

#ifdef DEBUG
                perror("urg marker begins read error");
#endif DEBUG

                        return READ_ERROR;
                    }
                    conversation->recv_mode =
                                    NI_EXPEDITED;
                }
            }
            if ((urg_mark != 1) &&
                (FD_ISSET(conversation->tcp_conv_id,
                                &read_template)))
```

```
{
    tcp_status =
            read(conversation->
                        tcp_conv_id,
              &(data[d_buf_ptr]),
                        sz_2_b_read);
    if (tcp_status > 0)
    {
        *data_len += tcp_status;
        if (sz_2_b_read == tcp_status)
        /*          required data       */
            if (conversation->recv_mode
                        == NI_EXPEDITED)
                return NI_URG_PENDING;
            else
                return OK;
        sz_2_b_read -= tcp_status;
        /*       read fewer bytes       */
        d_buf_ptr += tcp_status;
        /* advance data buffer pointer  */
    }
    else if (tcp_status == 0)   /* eof  */
    {
        return NI_EOT;
    }
    else if (errno != EWOULDBLOCK)
    {

#ifdef DEBUG
        perror("read");
```

```
#endif DEBUG

                              return READ_ERROR;
                          }
                      }
                  }
                  if (tcp_status < 0)
                  /*      some error in select                   */
                      return READ_ERROR;
                  /*          timer expired so  continue         */
              } while (TRUE);
          }
#endif TCP_SUPPORTED

#ifdef SNA_SUPPORTED
          if (conversation->network_type == NI_SNA)
          {
              /*              do something                       */
          }
#endif SNA_SUPPORTED

#ifdef ISO_SUPPORTED
          if (conversation->network_type == NI_OSI)
          {
              int asked_sz,
                  nxt,
                  recv_sz;
              int i,
                  j,
                  k;
```

```
asked_sz = *data_len;
j = nxt = 0;
j = recv_data_arr[Cv_id].size -
                recv_data_arr[Cv_id].curr_ptr;
if (j > asked_sz)
/* there is more data in  the buffer than asked */
    j = asked_sz;
if (j > 0)
/* some data already in the buffer              */
{
    bcopy(&(recv_data_arr[Cv_id].
            data[recv_data_arr[Cv_id].
                curr_ptr]), &(data[nxt]), j);
    recv_data_arr[Cv_id].curr_ptr += j;
    if ((recv_data_arr[Cv_id].curr_ptr) ==
        recv_data_arr[Cv_id].size)
    {
        free(recv_data_arr[Cv_id].data);
        recv_data_arr[Cv_id].curr_ptr = 0;
        recv_data_arr[Cv_id].size = 0;
        /* back to no reserve data in           */
        /* implementation buffer                */
        if (conversation->recv_mode ==
                                    NI_EXPEDITED)
        {
            conversation->recv_mode = NI_NORMAL;
            *data_len += j;
            return OK;
        }
```

```
            }
            asked_sz -= j;
            nxt += j;
            if (conversation->recv_mode == NI_EXPEDITED)
            {
                /*          more urgent data  left      */
                *data_len = nxt;
                return NI_URG_PENDING;
            }
        }


        while (asked_sz > 0)
        /* requested data is more than already buffered */
        /* or read                                      */
        {
            if (TReadRequest(conversation->tsap_conv_id,
                    tx_recv, NOTOK, td) == NOTOK)
            {
                int err;

#ifdef DEBUG
    printf("OSI error in read request 1, %s %s\n",
                TErrString(td->td_reason), td->td_data);
#endif DEBUG

                err = td->td_reason;
                free(td);
                *data_len = nxt;
                if (err == DR_NETWORK ||
```

```
                                err == DR_NORMAL)
                    {
                        free(recv_data_arr[Cv_id].data);
                        recv_data_arr[Cv_id].size = 0;
                        recv_data_arr[Cv_id].curr_ptr = 0;
                        return NI_EOT;
                    }
                    else
                        return READ_ERROR;
                }   /* TReadRequest == NOTOK */

#ifdef DEBUG
    printf(" OSI data received, size %d \n",
                                tx_recv->tx_cc);
#endif DEBUG

                if ((conversation->recv_mode == NI_NORMAL)
                                && (tx_recv->tx_expedited))
                {
                    rd_buf_q(recv_data_arr[Cv_id].data,
                        tx_recv->tx_qbuf.qb_forw);
                    recv_data_arr[Cv_id].size =
                                        tx_recv->tx_cc;
                    recv_data_arr[Cv_id].curr_ptr = 0;
                    TXFREE(tx_recv);
                    conversation->recv_mode = NI_EXPEDITED;
                    return NI_URG_PENDING;
                }

                if (asked_sz < tx_recv->tx_cc)
```

```
/* requested size is less than received     */
/* size so store the received data in buffer*/
{
    recv_data_arr[Cv_id].data =
        (char *) malloc(tx_recv->tx_cc);
    rd_buf_q(recv_data_arr[Cv_id].data,
        tx_recv->tx_qbuf.qb_forw);

    bcopy(recv_data_arr[Cv_id].data,
                &(data[nxt]), asked_sz);
    recv_data_arr[Cv_id].size =
                        tx_recv->tx_cc;
    recv_data_arr[Cv_id].curr_ptr = asked_sz;
    /* set pointer to first  unread byte    */
    nxt += asked_sz;
    asked_sz = 0;
    TXFREE(tx_recv);
}
else if (asked_sz == tx_recv->tx_cc)
/* requested size is equal to received size */
/* so read the received data into the user  */
/* data buffer                              */


{
    rd_buf_q(&(data[nxt]),
            tx_recv->tx_qbuf.qb_forw);
    nxt += asked_sz;
    asked_sz = 0;
    TXFREE(tx_recv);
```

```
                }
                else
                /* else requested size is greater than      */
                /* received size so increment the user data */
                /* pointer                                  */
                {
                    rd_buf_q(&(data[nxt]),
                                tx_recv->tx_qbuf.qb_forw);
                    nxt += tx_recv->tx_cc;
                    asked_sz -= tx_recv->tx_cc;
                    TXFREE(tx_recv);
                }

            }   /*              while                       */
            *data_len = nxt;
            free(td);
            return OK;
        }
#endif ISO_SUPPORTED
    }
    else
    {

#ifdef TCP_SUPPORTED
        if (conversation->network_type == NI_TCP)
        {

            FD_ZERO(&read_template);
            FD_SET(conversation->tcp_conv_id, &read_template);
```

```
tcp_status = select(FD_SETSIZE, &read_template,
    (fd_set *) 0, (fd_set *) 0, &wait);
/* wait should be a configuration parameter     */

if (tcp_status > 0)
{

    if (FD_ISSET(conversation->tcp_conv_id,
                                  &read_template))
    {
        tcp_status =
                read(conversation->tcp_conv_id,
                    data, *data_len);
        if (tcp_status > 0)
        {
            *data_len = tcp_status;
            /*          some data is read       */
            return OK;
        }
        else if (tcp_status == 0)   /*  eof     */
        {
            *data_len = 0;
            return NI_EOT;
        }
        if (errno == EWOULDBLOCK)
            return REPEAT_LATER;
        return READ_ERROR;
    }
}
if (tcp_status < 0)
```

```
                return READ_ERROR;  /* some error in select */
            return REPEAT_LATER;
            /*      else timer expired so repeat later      */
        }
#endif TCP_SUPPORTED

#ifdef SNA_SUPPORTED
        if (conversation->network_type == NI_SNA)
        {
        }
#endif SNA_SUPPORTED

#ifdef ISO_SUPPORTED
        if (conversation->network_type == NI_OSI)
        {
            int asked_sz,
                nxt,
                recv_sz;
            int i,
                j,
                k;

            asked_sz = *data_len;
            nxt = 0;
            j = recv_data_arr[Cv_id].size -
                            recv_data_arr[Cv_id].curr_ptr;
            if (j > asked_sz)
            /* there is more data in the buffer than asked  */
                j = asked_sz;
            if (j > 0)
```

```
/*      some data already in the buffer         */
{
    bcopy(&(recv_data_arr[Cv_id].
                data[recv_data_arr[Cv_id].
                curr_ptr]), &(data[nxt]), j);
    recv_data_arr[Cv_id].curr_ptr += j;
    if ((recv_data_arr[Cv_id].curr_ptr) ==
        recv_data_arr[Cv_id].size)
    {
        free(recv_data_arr[Cv_id].data);
        recv_data_arr[Cv_id].curr_ptr = 0;
        recv_data_arr[Cv_id].size = 0;
        /*  back to no reserve data in          */
        /*  implementation buffer               */
    }
    asked_sz -= j;
    nxt += j;
}


if (asked_sz > 0)
/* requested data is more than already buffered */
/* or read                                      */
{
    int nfds,
        status;
    fd_set mask;

    nfds = 0;
    FD_ZERO(&mask);
```

```
        if ((status =
                TSelectMask(
                    conversation->tsap_conv_id,
                    &mask, &nfds, td)) == NOTOK)
        {

#ifdef DEBUG
            printf("error in TSelect, %s, %s\n",
                TErrString(td->td_reason),
                td->td_data);
#endif DEBUG

            free(td);
            *data_len = nxt;
            return READ_ERROR;
        }

        if ((xselect(nfds, &mask,
                NULLFD, NULLFD, WAIT_SEC)) == NOTOK)
        {

#ifdef DEBUG
            printf("error in xselect, %s, %s\n",
                TErrString(td->td_reason),
                td->td_data);
#endif DEBUG

            free(td);
            *data_len = nxt;
            return READ_ERROR;
```

```
                }

                if (FD_ISSET(
                    conversation->tsap_conv_id, &mask) == 0)
                {
                    free(td);
                    *data_len = nxt;
                    if (*data_len)
                        return OK;
                    else
                        return REPEAT_LATER;
                }
                if (TReadRequest(conversation->tsap_conv_id,
                        tx_recv, OK, td) == NOTOK)
                {
                    int err;

#ifdef DEBUG
        printf("OSI error in read request 2, %s %s\n",
                            TErrString(td->td_reason),
                                        td->td_data);
#endif DEBUG

                    err = td->td_reason;
                    free(td);
                    *data_len = nxt;
                    if (err == DR_NETWORK ||
                                    err == DR_NORMAL)
                    {
                        free(recv_data_arr[Cv_id].data);
```

```
                recv_data_arr[Cv_id].size = 0;
                recv_data_arr[Cv_id].curr_ptr = 0;
                return NI_EOT;
            }
            else
                return READ_ERROR;
        }   /* TReadRequest == NOTOK */

#ifdef DEBUG
        printf(" OSI data received, size %d \n",
                                        tx_recv->tx_cc);
#endif DEBUG

        if (asked_sz < (tx_recv->tx_cc))
        /* requested size is less than received     */
        /* size so store the received data in buffer*/
        {
            recv_data_arr[Cv_id].data =
                (char *) malloc(tx_recv->tx_cc);
            rd_buf_q(recv_data_arr[Cv_id].data,
                tx_recv->tx_qbuf.qb_forw);

            bcopy(recv_data_arr[Cv_id].data,
                        &(data[nxt]), asked_sz);
            recv_data_arr[Cv_id].size =
                                tx_recv->tx_cc;
            recv_data_arr[Cv_id].curr_ptr = asked_sz;
            /* set pointer to first unread byte     */
            asked_sz = 0;
            nxt += asked_sz;
```

```
            TXFREE(tx_recv);
        }
        else if (asked_sz == (tx_recv->tx_cc))
        /* requested size is equal to received size */
        /* so read the received data into the user  */
        /* data buffer                              */
        {
            rd_buf_q(&(data[nxt]),
                            tx_recv->tx_qbuf.qb_forw);
            asked_sz = 0;
            nxt += asked_sz;
            TXFREE(tx_recv);
        }
        else
        /* else requested size is greater than recvd*/
        /* size so increment the  user data pointer */
        {
            rd_buf_q(&(data[nxt]),
                            tx_recv->tx_qbuf.qb_forw);
            nxt += tx_recv->tx_cc;
            asked_sz -= tx_recv->tx_cc;
            TXFREE(tx_recv);
        }
    }   /*          if asked_sz > 0                 */
    *data_len = nxt;
    /* data secured so far is the same as value     */
    /* of the nxt pointer                           */
    free(td);
    return OK;
}
```

```
#endif ISO_SUPPORTED
    }

}
```

```
/**********************************************************
*                         ewrite.c                        *
*                                                         *
*          Author Debashish Chatterjee                    *
*          Date   19th July 1990                          *
*                                                         *
*   int ewrite(Cv_id, blocking, priority, data , data_len)  *
*   "ewrite" sends data in a blocking or non-blocking mode  *
*   over a conversation                                   *
*   Input : The conversation identifier "Cv_id"           *
*          blocking or non-blocking mode "blocking"       *
*          "priority" is kind of data transfer expected. If *
*          priority is NI_EXPEDITED, this data transferred  *
*          as urgent. Only available with Blocking call.  *
*          Default is NI_NORMAL.                          *
*          pointer to the data buffer "data"              *
*   Input/output : "data_len" contains the size of the data  *
*          to be sent on input, and actual no of bytes    *
*          transmitted on return                          *
*                                                         *
*   Blocking Semantics:                                   *
*   If it returns OK then it was successful in reading in   *
*   some data.                                            *
*   If "data_len" is less than specified at input end of data*
*   transmission has been reached and data_len specifies the *
*   actual bytes read.                                    *
*   Else  there was internal problem in reading reflected by *
*   WRITE_ERROR.                                          *
*   Non-blocking semantics:                               *
*   If the mode is non-blocking and the call returns OK then *
```

```
*   some data was written and the actual size is specified by*
*   "data_len". If no data could be sent immediately,        *
*   REPEAT_LATER is returned. The caller should try again.   *
*   If NI_BAD_PRIORITY is returned, caller tried to send     *
*   urgent data in a non-blocking call or the agreed mode at *
*   the conversation establishment time was NI_NORMAL.       *
*                                                            *
*************************************************************/

/*  include files from the include directory                */
#include  "global.h"
#include  "shared.h"
```

```
/**********************************************************
*                     the ewrite function                 *
**********************************************************/

int ewrite(Cv_id, blocking, priority, data, data_len)
NA_conv Cv_id;  /* Conversation identifier                       */
int blocking;   /* Blocking or Non-blocking data write           */
int priority;   /* NI_NORMAL or NI_EXPEDITED                     */
char *data;     /* Pointer to Data buffer                        */
int *data_len;  /* Size of Data to be transmitted;on return */
                /* actual value transmitted                      */

{
    NA_conv_info *conversation;
    /* the conversation structure referring to Cv_id          */
#ifdef TCP_SUPPORTED
    int tcp_status;       /* status returned by TCP calls      */
    fd_set write_template;  /* socket write mask for           */
                            /*   multiplexing                  */
    struct timeval wait;    /* UNIX time structure             */
    int d_buf_ptr = 0;    /* pointer to the position in        */
                          /* the data buffer for subsequent    */
                          /* writes                            */
    int sz_2_rite;        /* size of data to be written        */
                          /* for that call                     */
#endif TCP_SUPPORTED
    /* Transmits data to partner on the connection            */
    /* established for it                                     */
    /* returns OK if write is successful else error code      */
```

```
#ifdef DEBUG
    printf("cv_id,data,datalen %d,%s,%d\n",
                                        Cv_id, data, *data_len);
#endif DEBUG
    if ((blocking == NON_BLOCKING)
                                  && (priority == NI_EXPEDITED))
        return NI_BAD_PRIORITY;
    if ((Cv_id < 1) || (Cv_id > MAX_CONVERSATIONS))
        return BAD_C_HANDLE;
    Cv_id--;
    if (conv_avail[Cv_id] == 0)
        return BAD_C_HANDLE;

    conversation = conv_arr[Cv_id];
    if (conversation->connected != NI_DONE)
        return NI_BAD_OPERATION;

    if ((conversation->priority == NI_NORMAL) &&
        (priority == NI_EXPEDITED))
    {
        return NI_BAD_PRIORITY;
        /* cannot use urgent if not declared at               */
        /* connection time or if partner does not accept      */
        /* urgent data                                        */
    }
    if ((priority == NI_EXPEDITED)
                                  && (*data_len > MAX_URG_SZ))
        return NI_BAD_OPERATION;
#ifdef TCP_SUPPORTED
    wait.tv_sec = WAIT_SEC;
```

```
    /* configuration parameter  defined in localconfig.h    */
    wait.tv_usec = WAIT_U_SEC;
    /* configuration parameter  defined in localconfig.h    */
#endif TCP_SUPPORTED


    if (blocking)       /* choose transmission mode         */
    {
#ifdef TCP_SUPPORTED
        if (conversation->network_type == NI_TCP)
        /*      is the network TCP/IP                       */
        {
            /* initialise the data len to be written        */
            sz_2_rite = *data_len;
            *data_len = 0;

            do
            {
                FD_ZERO(&write_template);
                FD_SET(conversation->tcp_conv_id,
                                            &write_template);

                tcp_status = select(FD_SETSIZE,
                                (fd_set *) 0, &write_template,
                                        (fd_set *) 0, &wait);
                if (tcp_status > 0)
                {
                    if (FD_ISSET
                            (conversation->tcp_conv_id,
                                        &write_template))
```

```
                {
                    if (priority == NI_EXPEDITED)
                    {
                        char c = 'y';
                        if ((d_buf_ptr == 0) &&
                            (conversation->send_mode
                                        == NI_NORMAL))
                        /* start of urgent data transfer*/
                        {
                            tcp_status =
                                send(conversation->
                                        tcp_conv_id,
                                        &c, 1, MSG_OOB);
                            if (tcp_status < 0)
                            {
#ifdef DEBUG
    perror("urg mark start error");
#endif DEBUG
                                return WRITE_ERROR;
                            }
#ifdef DEBUG
    printf("transmitted urg start marker\n");
#endif DEBUG
                            conversation->send_mode =
                                            NI_EXPEDITED;
                            continue;
                        }
                        else if ((sz_2_rite == 0) &&
                            (conversation->send_mode ==
                                            NI_EXPEDITED))
```

```
                                        /*  end of urgent data transfer */
                                        {
                                            tcp_status =
                                                    send(conversation->
                                                            tcp_conv_id,
                                                       &c, 1, MSG_OOB);
                                            if (tcp_status < 0)
                                            {
#ifdef DEBUG
    perror("urg mark end error");
#endif DEBUG
                                                return WRITE_ERROR;
                                            }
#ifdef DEBUG
    printf("transmitted urg end marker\n");
#endif DEBUG
                                            conversation->send_mode =
                                                                NI_NORMAL;
                                            /* toggle priority to normal*/
                                            return OK;
                                        }
                                    }
                                    tcp_status =
                                            write(conversation->
                                                    tcp_conv_id,
                                                    &(data[d_buf_ptr]),
                                                    sz_2_rite);
                                    /* standard TCP call for data send  */
                                    if (tcp_status > 0)
                                    {
```

```
                        *data_len += tcp_status;
                        if ((sz_2_rite == tcp_status)
                            && (conversation->send_mode
                                            == NI_NORMAL))
                            return OK;
                        /* required data written        */
                        sz_2_rite -= tcp_status;
                        /* transmit remaining bytes     */
                        d_buf_ptr += tcp_status;
                        /* advance data buf ptr         */
                    }
                    else if (errno != EWOULDBLOCK)
                    {
                        perror("read");
                        return WRITE_ERROR;
                    }
                }
                else
                    return WRITE_ERROR;
            }
            if (tcp_status < 0)
            /* select failed for some reason            */
                return WRITE_ERROR;
            /* else timer expired, keep trying          */
        } while (TRUE);
        /* repeat this loop until all the data is sent  */
    }
#endif TCP_SUPPORTED
#ifdef SNA_SUPPORTED
    if (conversation->network_type == NI_SNA)
```

```
        {
            /*          do something                              */
        }
#endif SNA_SUPPORTED
#ifdef ISO_SUPPORTED
        if (conversation->network_type == NI_OSI)
        {
            int more_tx,
                tx_sz,
                nxt;

            struct TSAPdisconnect *td;
            td = (struct TSAPdisconnect *)
                malloc(sizeof(struct TSAPdisconnect));
            more_tx = *data_len;
            nxt = 0;
            do
            {
                if (more_tx > 65400)
                {
                    tx_sz = 65400;
                    more_tx -= 65400;
                }
                else
                {
                    tx_sz = more_tx;
                    more_tx = 0;
                }
                if (priority == NI_EXPEDITED)
                {
```

```
                    if (TExpdRequest
                            (conversation->tsap_conv_id,
                            &(data[nxt]), tx_sz, td)
                                                == NOTOK)
                    {
#ifdef DEBUG
    printf("OSI could not transmit expedited data, %s %s\n",
                    TErrString(td->td_reason), td->td_data);
#endif DEBUG
                        free(td);
                        *data_len -= (more_tx + tx_sz);
                        return WRITE_ERROR;
                    }
                }
                else
                {
                    if (TDataRequest
                            (conversation->tsap_conv_id,
                            &(data[nxt]), tx_sz, td) ==
                                                    NOTOK)
                    {
#ifdef DEBUG
    printf("OSI could not transmit data, %s %s\n",
                TErrString(td->td_reason), td->td_data);
#endif DEBUG
                        free(td);
                        *data_len -= (more_tx + tx_sz);
                        return WRITE_ERROR;
                    }
                }
```

```
                nxt += tx_sz;
            } while (more_tx > 0);
            return OK;
        }
#endif ISO_SUPPORTED
        /* this loop is necessary because the communication */
        /* is non-blocking at this level                    */
    }
    else
    {
#ifdef TCP_SUPPORTED
        if (conversation->network_type == NI_TCP)
        {
            FD_ZERO(&write_template);
            FD_SET(conversation->tcp_conv_id,
                                        &write_template);

            tcp_status = select(FD_SETSIZE,
                                (fd_set *) 0, &write_template,
                                (fd_set *) 0, &wait);
            if (tcp_status > 0)
            {
                if (FD_ISSET(conversation->tcp_conv_id,
                                            &write_template))
                {
                    tcp_status =
                        write(conversation->tcp_conv_id,
                                            data, *data_len);
                    if (tcp_status > 0)
                    {
```

```
                    *data_len = tcp_status;
                    /* some data could be written        */
                    return OK;
                }
                if (errno == EWOULDBLOCK)
                    return REPEAT_LATER;
                /* no data could be written              */
                return WRITE_ERROR;
            }
            else
                return WRITE_ERROR;
        }
        if (tcp_status == 0)
            return REPEAT_LATER;
        /* timer expired and no data could be written    */
        return WRITE_ERROR;
        /* some other error, invalid time etc            */
    }
#endif TCP_SUPPORTED
#ifdef SNA_SUPPORTED
    if (conversation->network_type == NI_SNA)
    {
    }
#endif SNA_SUPPORTED
#ifdef ISO_SUPPORTED
    if (conversation->network_type == NI_OSI)
    {
        int more_tx,
            tx_sz;
        struct TSAPdisconnect *td;
```

```
            int nfds;
            fd_set mask;
            td = (struct TSAPdisconnect *)
                malloc(sizeof(struct TSAPdisconnect));
            if (*data_len > 65400)
                *data_len = 65400;

            nfds = 0;
            FD_ZERO(&mask);
            if (TSelectMask(conversation->tsap_conv_id,
                    &mask, &nfds, td) == NOTOK)
            {
#ifdef DEBUG
    printf("error in TSelect, %s, %s\n",
                    TErrString(td->td_reason),
                                        td->td_data);
#endif DEBUG
                free(td);
                *data_len = 0;
                return WRITE_ERROR;
            }

            if (xselect(nfds, NULLFD, &mask,
                            NULLFD, WAIT_SEC) == NOTOK)
            {
#ifdef DEBUG
    printf("error in xselect, %s, %s\n",
                    TErrString(td->td_reason),
                                    td->td_data);
#endif DEBUG
```

```
                        free(td);
                        *data_len = 0;
                        return WRITE_ERROR;
                    }

                    if (FD_ISSET(conversation->tsap_conv_id,
                                                        &mask) == 0)
                    {
                        free(td);
                        *data_len = 0;
                        return REPEAT_LATER;
                    }

                    if (TDataRequest(conversation->tsap_conv_id,
                                    data, *data_len, td) == NOTOK)
                    {
#ifdef DEBUG
                printf("OSI could not transmit data, %s %s\n",
                            TErrString(td->td_reason, td->td_data));
#endif DEBUG
                        free(td);
                        return WRITE_ERROR;
                    }
                    return OK;
                }
#endif ISO_SUPPORTED
    }

}
```

```
/************************************************************
*                       read_mpx.c                          *
*                                                           *
*           Author Debashish Chatterjee                     *
*           Date   19th July 1990                           *
*                                                           *
*   "rselect" selects conversations for reading data        *
*   Input: An array of conversation identifiers "arr_conv_id"*
*          Timeout "wait" in seconds                        *
*   Output: An array of integers of size "max_elem" to      *
*          indicate the selected conversations. Each element *
*          if 1 indicates that the corresponding conversation*
*          is selected.                                     *
*   Input/Output: On input "max_elem" contains size of the  *
*          above array. On output it contains total number of*
*          selected conversations                           *
*   Returns OK if successfully returned else returns        *
*   R_SELECT_FAILED.                                        *
*                                                           *
*                                                           *
************************************************************/


/* include files from the include directory                */
#include  "global.h"
#include  "shared.h"
```

```
/***********************************************************
*                    the rselect function                  *
***********************************************************/

int rselect(arr_conv_id, max_elem, arr_selected, wait)
NA_conv arr_conv_id[];  /* array of conversation identifiers*/
int *max_elem;          /* number of elements in the array  */
int arr_selected[];     /* array which returns the selected */
                        /* conversations                    */
int wait;               /* time in seconds to wait for check*/
{

#ifdef TCP_SUPPORTED
    fd_set read_template;
    struct timeval timeout;
#endif TCP_SUPPORTED

    int i,
        j,
        selected,
        status,
        curr;
    int tcp_set,
        sna_set,
        osi_set;

#ifdef ISO_SUPPORTED
    fd_set rmask;
    int nfds;
    struct TSAPdisconnect *td;
```

```
#endif ISO_SUPPORTED

    /*                  initialize                          */
    for (i = 0; i < *max_elem; i++)
        arr_selected[i] = 0;
    tcp_set = sna_set = osi_set = FALSE;

#ifdef TCP_SUPPORTED
    timeout.tv_sec = wait;
    timeout.tv_usec = 0;
#endif TCP_SUPPORTED

#ifdef ISO_SUPPORTED
    td = (struct TSAPdisconnect *)
                malloc(sizeof(struct TSAPdisconnect));
#endif ISO_SUPPORTED
    /*              end of initialize                       */

    /*      first set the masks for selecting               */
    for (i = 0, j = 0; i < *max_elem; i++)
    {
        if (arr_conv_id[i] < 1 ||
                        arr_conv_id[i] > MAX_CONVERSATIONS)
            continue;
        /*      Ignore silently improper conversations      */

        curr = arr_conv_id[i] - 1;

#ifdef TCP_SUPPORTED
        if (conv_arr[curr]->network_type == NI_TCP)
```

```
        {
            if (!tcp_set)
            {
                tcp_set = TRUE;
                FD_ZERO(&read_template);
                /* set a condition to indicate at least one */
                /* conversation is of TCP type and zero the */
                /* mask for TCP select call                 */
            }
            FD_SET(conv_arr[curr]->tcp_conv_id,
                                         &read_template);
            /* standard set call prior to select in TCP     */
        }
#endif TCP_SUPPORTED

#ifdef SNA_SUPPORTED
        if (conv_arr[curr]->network_type == NI_SNA)
        {
            if (!sna_set)
                sna_set = TRUE;
        }
#endif SNA_SUPPORTED

#ifdef ISO_SUPPORTED
        if (conv_arr[curr]->network_type == NI_OSI)
        {
            if (!osi_set)
            {
                osi_set = TRUE;
                FD_ZERO(&rmask);
```

```
                /* set a condition to indicate at least one */
                /* conversation is of OSI type and zero     */
                /* the mask for OSI xselect call            */
            }
            if ((status =
                    TSelectMask(conv_arr[curr]->tsap_conv_id,
                                &rmask, &nfds, td)) == NOTOK)
            {
#ifdef DEBUG
                printf("error in TSelect, %s, %s\n",
                        TErrString(td->td_reason),
                                    td->td_data);
#endif DEBUG
                free(td);
                return R_SELECT_FAILED;
            }
        }
#endif ISO_SUPPORTED
    }

    selected = 0;
    /* number of selected conversations out of max_elem     */

    /* now select the calls using protocol specific calls   */

#ifdef TCP_SUPPORTED
    if (tcp_set)
    /* if any of the conversations is of tcp type           */
    {
```

```
        status = select(FD_SETSIZE, &read_template,
                        (fd_set *) 0, (fd_set *) 0, &timeout);
        /*              standard tcp call                      */
        if (status > 0)
        {
            /*              select was ok                      */
            for (i = 0; i < *max_elem; i++)
            {
                if (arr_conv_id[i] < 1 ||
                    arr_conv_id[i] > MAX_CONVERSATIONS)
                    continue;
                curr = arr_conv_id[i] - 1;
                if ((conv_arr[curr]->network_type == NI_TCP)
                    && (FD_ISSET(conv_arr[curr]->tcp_conv_id,
                                            &read_template)))
                {
                    arr_selected[i] = 1;
/* set the corresponding index to 1     */
                    selected++;
                }
            }
        }
        if (status < 0)
            return R_SELECT_FAILED;
    }
#endif TCP_SUPPORTED

#ifdef SNA_SUPPORTED
    /*                  similarly for sna                      */
#endif SNA_SUPPORTED
```

```
#ifdef ISO_SUPPORTED
    if (osi_set)
    {
        if ((xselect(nfds, &rmask, NULLFD, NULLFD, wait))
                                                    == NOTOK)
        {
#ifdef DEBUG
            printf("error in xselect, %s, %s\n",
                            TErrString(td->td_reason),
                                        td->td_data);
#endif DEBUG
            free(td);
            return R_SELECT_FAILED;
        }
        for (i = 0; i < *max_elem; i++)
        {
            if (arr_conv_id[i] < 1 ||
                arr_conv_id[i] > MAX_CONVERSATIONS)
                continue;
            curr = arr_conv_id[i] - 1;
            if ((conv_arr[curr]->network_type == NI_OSI) &&
                (FD_ISSET(conv_arr[curr]->tsap_conv_id,
                                                &rmask)))
            {
                arr_selected[i] = 1;
                selected++;
            }
        }
    }           /* end of osi specific multiplex calls      */
```

```
#endif ISO_SUPPORTED
    *max_elem = selected;
    if (selected == 0)
        return TIMED_OUT;

    return OK;

}                /*              rselect                      */
```

```
/***********************************************************
*                    write_mpx.c                           *
*                                                          *
*   "wselect" selects conversations for sending data       *
*   Input: An array of conversation identifiers "arr_conv_id"*
*          Timeout "wait" in seconds                       *
*   Output: An array of integers of size "max_elem" to     *
*          indicate the selected conversations. Each element *
*          if 1 indicates that the corresponding conversation*
*          is selected.                                    *
*   Input/Output: On input "max_elem" contains size of the   *
*          above array. On output it contains total number of*
*          selected conversations                          *
*   Returns OK if successfully returned else returns       *
*   W_SELECT_FAILED.                                       *
*                                                          *
*                                                          *
***********************************************************/

/* include files from the include directory              */
#include  "global.h"
#include "shared.h"
```

```
/************************************************************
*                      the wselect function                 *
************************************************************/
int wselect(arr_conv_id, max_elem, arr_selected, wait)
NA_conv arr_conv_id[];  /* array of conversation identifiers*/
int *max_elem;          /* number of elements in the array  */
int arr_selected[];     /* array which returns the selected */
                        /* conversations                    */
int wait;               /* time in seconds to wait for check*/
{

#ifdef TCP_SUPPORTED
    fd_set write_template;
    struct timeval timeout;
#endif TCP_SUPPORTED

    int i,
        j,
        selected,
        status,
        curr;
    int tcp_set,
        sna_set,
        osi_set;

#ifdef ISO_SUPPORTED
    fd_set wmask;
    int nfds;
    struct TSAPdisconnect *td;
```

```
#endif ISO_SUPPORTED

    /*                   initialize                          */
    for (i = 0; i < *max_elem; i++)
        arr_selected[i] = 0;
    tcp_set = sna_set = osi_set = FALSE;

#ifdef TCP_SUPPORTED
    timeout.tv_sec = wait;
    timeout.tv_usec = 0;
#endif TCP_SUPPORTED

#ifdef ISO_SUPPORTED
    td = (struct TSAPdisconnect *)
            malloc(sizeof(struct TSAPdisconnect));
#endif ISO_SUPPORTED
    /*              end of initialize                        */

    /*      first set the masks for selecting                */
    for (i = 0, j = 0; i < *max_elem; i++)
    {
        if (arr_conv_id[i] < 1 ||
                arr_conv_id[i] > MAX_CONVERSATIONS)
            continue;
        /*      Ignore silently improper conversations       */

        curr = arr_conv_id[i] - 1;
#ifdef TCP_SUPPORTED
        if (conv_arr[curr]->network_type == NI_TCP)
        {
```

```
            if (!tcp_set)
            {
                tcp_set = TRUE;
                FD_ZERO(&write_template);
                /* set a condition to indicate at least one */
                /* conversation is of TCP type and zero the */
                /* mask for TCP select call                 */
            }
            FD_SET(conv_arr[curr]->tcp_conv_id,
                                        &write_template);
            /* standard set call prior to select in TCP     */
        }
#endif TCP_SUPPORTED

#ifdef SNA_SUPPORTED
        if (conv_arr[curr]->network_type == NI_SNA)
        {
            if (!sna_set)
                sna_set = TRUE;
        }
#endif SNA_SUPPORTED

#ifdef ISO_SUPPORTED
        if (conv_arr[curr]->network_type == NI_OSI)
        {
            if (!osi_set)
            {
                osi_set = TRUE;
                FD_ZERO(&wmask);
                /* set a condition to indicate at least one */
```

```
                /* conversation is of OSI type and zero     */
                /* the mask for OSI xselect call            */
            }
            if ((status =
                    TSelectMask(conv_arr[curr]->tsap_conv_id,
                                &wmask, &nfds, td)) == NOTOK)
            {
#ifdef DEBUG
                printf("error in TSelect, %s, %s\n",
                            TErrString(td->td_reason),
                                            td->td_data);
#endif DEBUG
                free(td);
                return W_SELECT_FAILED;
            }
        }
#endif ISO_SUPPORTED
    }

    selected = 0;
    /* number of selected conversations out of max_elem     */

    /* now select the calls using protocol specific calls   */

#ifdef TCP_SUPPORTED
    if (tcp_set)
    /* if any of the conversations is of tcp type            */
    {

        status = select(FD_SETSIZE, (fd_set *) 0,
```

```
                        &write_template, (fd_set *) 0,
                        &timeout);
        /*          standard tcp call                        */
        if (status > 0)
        {
            /*              select was ok                    */
            for (i = 0; i < *max_elem; i++)
            {
                if (arr_conv_id[i] < 1 ||
                    arr_conv_id[i] > MAX_CONVERSATIONS)
                    continue;
                curr = arr_conv_id[i] - 1;
                if ((conv_arr[curr]->network_type == NI_TCP)
                    && (FD_ISSET(conv_arr[curr]->tcp_conv_id,
                                        &write_template)))
                {
                    arr_selected[i] = 1;
                    /* set the corresponding index to 1      */
                    selected++;
                }
            }
        }
        if (status < 0)
            return W_SELECT_FAILED;
    }
#endif TCP_SUPPORTED
#ifdef SNA_SUPPORTED
    /*                  similarly for sna                    */
#endif SNA_SUPPORTED
#ifdef ISO_SUPPORTED
```

```
    if (osi_set)
    {
        if ((xselect(nfds, NULLFD, &wmask, NULLFD, wait))
                                                    == NOTOK)
        {
#ifdef DEBUG
            printf("error in xselect, %s, %s\n",
                            TErrString(td->td_reason),
                                    td->td_data);
#endif DEBUG
            free(td);
            return W_SELECT_FAILED;
        }
        for (i = 0; i < *max_elem; i++)
        {
            if (arr_conv_id[i] < 1 ||
                arr_conv_id[i] > MAX_CONVERSATIONS)
                continue;
            curr = arr_conv_id[i] - 1;
            if ((conv_arr[curr]->network_type == NI_OSI) &&
                (FD_ISSET(conv_arr[curr]->tsap_conv_id,
                                            &wmask)))
            {
                arr_selected[i] = 1;
                selected++;
            }
        }
    }           /* end of osi specific multiplex calls      */
#endif ISO_SUPPORTED
    *max_elem = selected;
```

```
    if (selected == 0)
        return TIMED_OUT;

    return OK;

}               /*              wselect                         */
```

## E.4 Utilities

```
/************************************************************
*                        util.c                             *
*                                                           *
*           Author Debashish Chatterjee                     *
*           Date   19th July 1990                           *
*                                                           *
*          contains all the utility routines                *
*                                                           *
************************************************************/

/*           define the include files                       */
#include "global.h"
#include "shared.h"

/************************************************************
*                     mark_conv(Cv_id)                      *
*   mark a conversation identifier. Ignore if already marked.*
************************************************************/
void mark_conv(Cv_id)
NA_conv *Cv_id;
{
#ifdef DEBUG
    printf("util.c : conversation to be marked is %d\n",
                                                    *Cv_id);
#endif DEBUG
    /* Conversation identifiers are always between 1 and    */
    /* MAX_CONVERSATIONS. Ignore beyond this range          */
    if (*Cv_id < 1 || *Cv_id > MAX_CONVERSATIONS)
        return;
```

```
    /* Conversation identifier is in range                          */
    *Cv_id = -(*Cv_id);
    /* negate the conversation identifier                           */
#ifdef DEBUG
    printf("util.c : marked conversation as %d\n", *Cv_id);
#endif DEBUG
    return;

}
```

```
/************************************************************
*                         unmark_conv(Cv_id)                *
*   unmark a conversation identifier if negative and between *
*  -1 and  -MAX_CONVERSATIONS. Else ignore.                 *
************************************************************/

void unmark_conv(Cv_id)
NA_conv *Cv_id;
{
#ifdef DEBUG
    printf("util.c : conversation to be unmarked is %d\n",
                                                    *Cv_id);
#endif DEBUG
    /* Marked conversation identifiers are always between
     * -1 and -MAX_CONVERSATIONS. Ignore beyond this
     * range */

    if (*Cv_id < -MAX_CONVERSATIONS || *Cv_id > -1)
        return;

    /* Marked conversation identifier is in range */
    *Cv_id = -(*Cv_id); /* negate the conversation
                 * identifier to unmark it */
#ifdef DEBUG
    printf("util.c : unmarked conversation as %d\n", *Cv_id);
#endif DEBUG
    return;

}
```

```
/***********************************************************
*                       ni_init                            *
*   initialises if threads are being used                  *
***********************************************************/
#ifdef CTHREADS_SUPPORTED
void ni_init()
{
    na_rsrc_lock = mutex_alloc(); /*initialize resource lock*/
    na_conv_lock = mutex_alloc(); /*same for conversation   */
}
#endif CTHREADS_SUPPORTED
```