UNIVERSITY OF SOUTHAMPTON

DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

THE COMMUNICATION ROUTINES — A NETWORK LAYER COMMUNICATION MODEL

Jonathan Dale

University of Southampton

Technical Report No. 95-2

January 1995

ISBN: 0-854-325-786

The Communication Routines - A Network Layer Communication Model

Jonathan Dale

Multimedia Research Group

Department of Electronics and Computer Science

University of Southampton

UK

jd94r@ecs.soton.ac.uk

1. Introduction

The Communication Routines are a small set of C function calls that provide *stream* (reliable, connection-oriented) and *datagram* (best-try, connection-less) communication within the Transmission Control Protocol/Internet Protocol (TCP/IP) communication stack. They are based around the Inter-Process Communication (IPC) mechanism called *sockets*, which provide both stream and datagram oriented services.

The design criteria for the Communication Routines were based upon the following considerations:

- To provide a suitable application interface to the sockets IPC which would allow a certain level of abstraction between the networking layer (TCP/IP and sockets) and the application layer (client-server software).
- To allow client-server software to be built rapidly and easily.
- To be available across UNIX and Winsock compatible (PC) platforms.

2. Network Communication

In the TCP/IP networking environment, communication generally takes place between *clients* and *servers*, hence the term *client-server relationships*. A server is a network entity which provides a set of services to a range of clients. Clients can connect to various servers to take advantage of services, which the servers undertake on their behalf. It follows, then, that some form of message exchange (that is, communication) must take place between the client and the server, and vice-versa.

TCP/IP supports two essential modes of network communication:

- Streams are a connection-oriented form of communication, which means that there is a persistent connection between the client and the server for the duration of the communication (rather like a telephone conversation). Figure 1 illustrates that the client and the server are *synchronised* during their communication.
- Datagrams are a connection-less form of communication, which means that there is not a persistent connection between the client and the server. Each message between the client and the server is a separate connection, containing its own addressing information (this is analogous to the postal mail service). With datagram client-server relationships, the client is not tied to the server for the duration of the request, nor is the server tied to the client after is has completed the work for that client; they are both said to exist *asynchronously*. The essential difference between stream and datagram communication is that streams are persistent for the *entire* communication and require set-up, whereas datagrams are transient and do not require any set-up. This is shown in figure 2.

A third communication exists, called *broadcasting*, which is a variation of the datagram communication. Broadcasting allows a client or a server send the same message to every network entity within the local network (called the *subnet*). This can be useful for a number of reasons, for example, propagating common data to an entire network, searching for a particular service within the network, etc. Each network entity receives the broadcasted message and either passes it onto an appropriate software

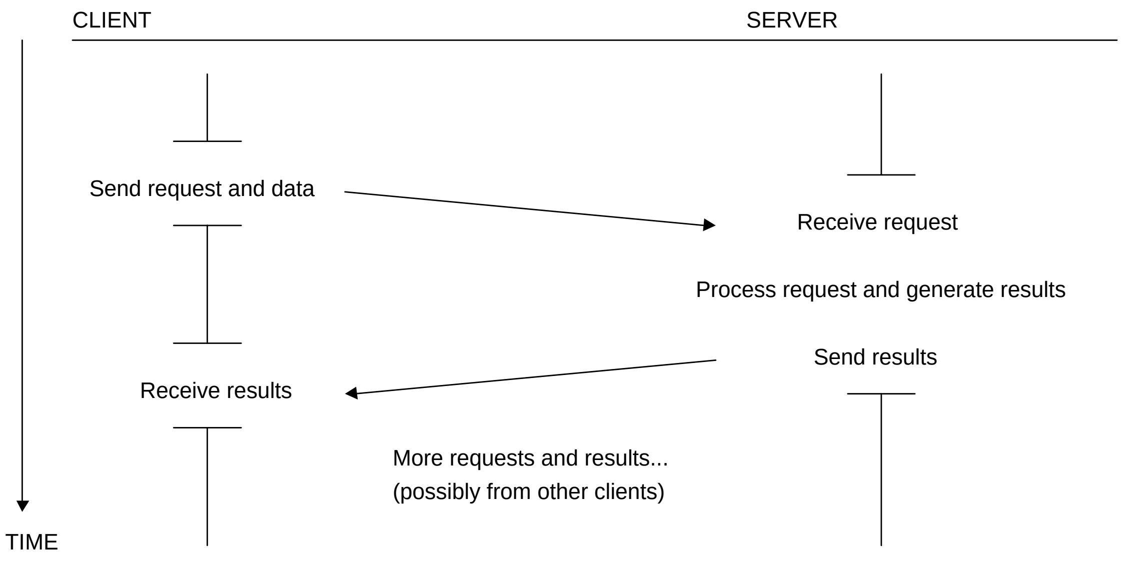
CLIENT
SERVER
Send request and data
Receive request
Process request and generate results
Send results
Receive results
More requests and results...
(possibly from other clients)
TIME

Figure 1 : Client-Server Communication Pattern in a Stream Communication

entity which it is running or discards it as unwanted. Broadcasting, however, should be kept to a minimum since it can flood the network and reduce performance.

For datagram and broadcast communication channels, the maximum amount of data that can be transmitted at a time is 8192 bytes. For streams, any amount of data can be written to the communication channel but it is the responsibility of the receiving network entity to read the data in a sensible fashion.

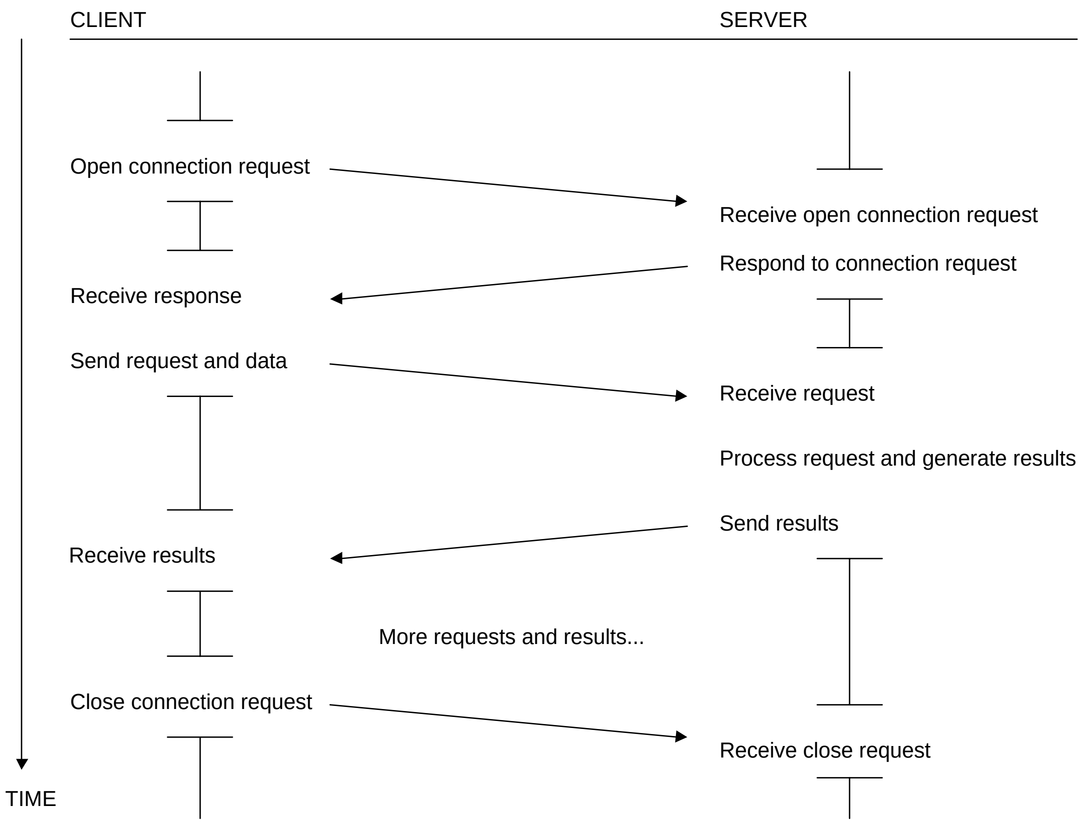

Figure 2 : Client-Server Communication Pattern in a Datagram Communication

3. Communication Handles

From the point of view of client and server software using the Communication Routines, the key element to network communication is the network communication handle, or just *handle*, which is a pointer to a communication session. Before any communication can take place between clients and servers, each needs to *register* for a communication session, the semantics of which are determined by the communication type selected (stream, datagram or broadcast).

For a stream connection, the roles of the client and the server are well defined; the client initiates a connection with the server, an amount of two-way dialogue takes place and then the client closes the connection. The client is always a client and the server is always a server. This means that during registration, the server must register its session first (getting a *primary handle*) and then, when the client registers its session, the server *accepts* this connection (getting a *secondary handle*). The primary handle is used to allow the server to offer a server (to say "Here I am") and the secondary handle is used to actually allow the client and server to communicate. Secondary handles are unaccepted when the communication between the client and the server has finished, and primary handles are unregistered when the server wishes to terminate.

This is demonstrated by the code fragment given in table 1 (most of the function parameters have been omitted for simplicity).

CLIENT	SERVER
	/* Make the server available */ `PrimaryHandle = CR_RegisterServer;`
/* The client registers with the server */ `Handle = CR_RegisterClient;`	/* The server accepts this registration */ `SndaryHandle = CR_Accept;`
/* The client sends some data */ `CR_Send (Handle, some_data);`	/* The server receives some data */ `CR_Receive (SndaryHandle, some_data);`
/* The client receives the results */ CR_Receive (Handle, some_results);	/* The server sends the results */ `CR_Send (SndaryHandle, some_results);`
... etc ...	... etc ...
/* The client ends the communication */ `CR_Unregister (Handle);`	/* The server unaccepts from the client */ `CR_Unaccept (SndaryHandle);`
/* The client can connect to another server */	/* The server can accept a connection from a another client */

Table 1 : Client-Server Function Calls in a Stream Communication

However, with datagram (and broadcast) communication, a server is classed as a server when it receives data from a client. When a server needs to send data back to a client, it must temporarily become a client and register a new communication session. Thus, the terms client and server in datagram communication depend upon what action the network entity is performing.

The corresponding functions calls for a datagram connection are given in table 2.

CLIENT	SERVER
	/* Make the server available */ `RecvHandle = CR_RegisterServer;`
/* The client registers for communication */ `SendHandle = CR_RegisterClient;`	
/* The client sends some data */ `CR_Send (SendHandle, some_data);`	/* The server receives some data */ `CR_Receive (RecvHandle, some_data);`
/* The client becomes a server to receive the results */ `RecvHandle = CR_RegisterServer;`	/* The server becomes a client to return the result */ `SendHandle = CR_RegisterClient;`
/* The client receives the result */ `CR_Receive (RecvHandle, some_result);`	/* The server sends the result */ `CR_Send (SendHandle, some_result);`
/* The client reverts back to a client */ `CR_Unregister (RecvHandle);`	/* The server reverts back to a server */ `CR_Unregister (SendHandle);`
/* The client can now send data to other servers */	/* The server can receive data from other clients */

Table 2 : Client-Server Function Calls in a Datagram Communication

4. Network Abstraction

The Communication Routines provide a layer of network abstraction above the transport layer of the TCP/IP protocol stack (figure 3) which affords many advantages to the network programmer when compared with standard socket programming:

- Network programming is simplified since most of the network structures and functions are hidden behind the Application Programming Interface (API). An application needs only to know the network address of another network entity to make a connection and only needs to quote the handle of the communication channel to be able to send or receive data. This information hiding allows the programmer to concentrate on writing the application and not on the intricacies of network programming. Additionally, the task of porting the code to another network stack is made easier since it only means rewriting the Communication Routines for the new protocol and altering the network address naming in the application.
- Time-out control is provided for reading from and writing to communication channels. In order to provide resilient and fault-tolerant systems, it is important to introduce the concept of deadlines, or time-outs, so that an application can not only determine if systems are busy or unavailable, but also if and when an error has occurred. The time-out controls for reading and writing within the Communication Routines permit an application to be sure that it has either sent or received all of its data. If an error did occur, then the application knows how far through the communication it was and can either resend the remaining portion of data or request a retransmission.
- Service selection is available through a single API. With sockets programming, there are different function calls for different services, for example, streams use some different and extra functions than datagrams for the same operations. The Communication Routines unify network programming into a single and consistent API that supports all services that are available in the TCP/IP protocol stack.
- The shared libraries are available on UNIX and Winsock compatible platforms. The Communication Routines are written using ANSI-compliant C, which means that they can be compiled under any

platform that supports ANSI C and networking sockets.

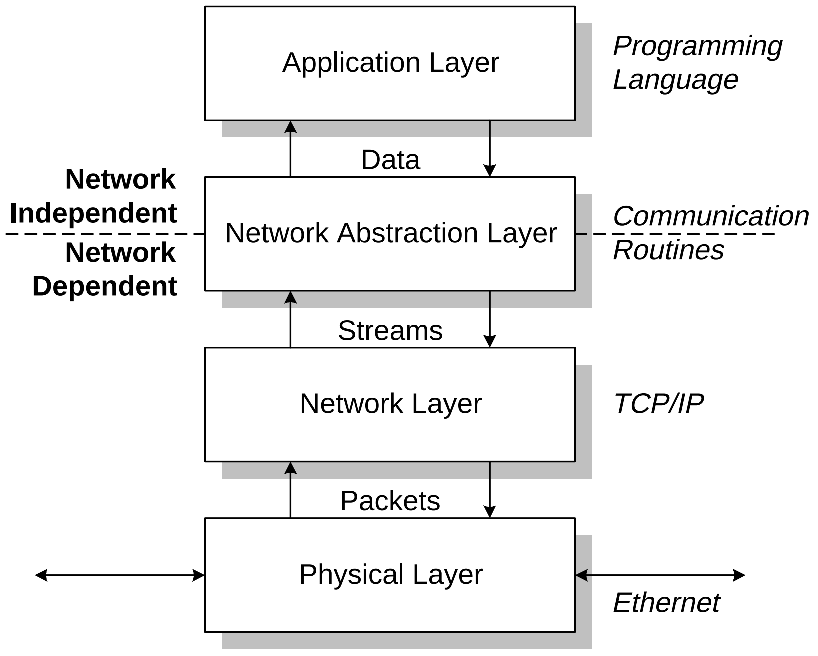

Figure 3 : Network Abstraction through an Intermediate Layer

5. Function Reference

5.1. API Summary

1. int CR_RegisterServer (int *nLocalPort*, int *nConnectionType*)
   Registers a server-style communication channel.
2. int CR_RegisterClient (LPSTR *szRemoteIPAddress*, int *nRemotePort*, int *nConnectionType*)
   Registers a client-style communication channel.
3. int CR_Accept (int *nReceivingHandle*)
   Completes a client-server stream communication channel.
4. INT CR_Send (int *nHandle*, LPHSTR *lphszData*, INT *IBytesToSend*, int *nTimeout*)
   Sends data to a communication channel.
5. INT CR_Receive (int *nHandle*, LPHSTR *lphszData*, INT *IBytesToRead*)
   Reads data from a communication channel.
6. INT CR_ReceiveN (int *nHandle*, LPHSTR *lphszData*, INT *IBytesToRead*, int *nTimeout*)
   Attempts to read data from a communication channel until all of the required amount of bytes have been read or the timeout has been reached.
7. int CR_GetInfo (int *nHandle*, LPINT *lpiPort*, LPSTR *lpszIPAddress*, int *nInfoType*)
   Obtains information about the local or remote machine attached to a given communication channel.
8. int CR_GetIPAddress (LPSTR *lpszHostName*, LPSTR *lpszIPAddress*)
   Obtains the Internet address for a given hostname.
9. int CR_GetLastError (void)
   Returns the error code associated with the last communication routine error that occurred.
10. int CR_Unaccept (int *nSecondaryHandle*)
    Disassociates the server from a client in a stream communication channel.
11. int CR_Unregister (int *nPrimaryHandle*)

Unregisters a communication channel.

12. int CR_UnregisterAll (void)
    Unregisters all registered communication channels.

5.2. int CR_RegisterServer (int nLocalPort, int nConnectionType)

Purpose
Registers a server-style connection for either stream or datagram communication. A server-style connection is characterised by the fact that the server typically 'listens' on the specified port, awaiting data from a client.

Parameters
nLocalPort
The local port number to use for the connection. If this value is 0, then any local port is used (use *CR_GetInfo* to determine which port has been chosen).
nConnectionType
The type of connection to make. It can be either *STREAM* for a duplex virtual circuit connection, *DATAGRAM* for a datagram connection, or *BROADCAST* for a datagram broadcasting connection.

Return values
The primary handle onto the communication channel. If there was a communication problem, then *COMM_ERROR* is returned, else if there was a problem adding the registration details, then *LINK_ERROR* is returned.

Error codes
CECANTBCAST
A server cannot make a broadcast connection.
LECANTADDHANDLE
The handle associated with this registration could not be created.

See also
CR_Send; *CR_Receive*; *CR_GetInfo*; *CR_Unregister*

5.3. int CR_RegisterClient (LPSTR szRemoteIPAddress, int nRemotePort, int nConnectionType)

Purpose
Registers a client-style connection for either stream or datagram communication. A client-style connection is characterised by the fact that the client typically writes data through a port to a waiting server, hence the need to specify a remote Internet address and port number.

Parameters
szRemoteIPAddress
The remote Internet address of the server.
nRemotePort
The remote port number of the server.
nConnectionType
The type of connection to make. It can be either *STREAM* for a duplex virtual circuit connection, *DATAGRAM* for a datagram connection, or *BROADCAST* for a datagram broadcasting connection.

Return values
The primary handle onto the communication channel. If there was a communication problem, then *COMM_ERROR* is returned, else if there was a problem adding the registration details, then *LINK_ERROR* is returned.

Error codes
CECANTCONNECT
The client could not connect correctly with the specified server.
LECANTADDHANDLE
The handle asscoaited with this registration could not be created.
LECANTALLOCMEM

Memory allocation failed.
LECANTADDINFO
The registration information associated with this communication channel could not be added.

Notes
CR_RegisterClient allocates any local port number (use *CR_GetInfo* to determine which port has been chosen).

See also
CR_Send; *CR_Receive*; *CR_GetInfo*; *CR_Unregister*

5.4. int CR_Accept (int nReceivingHandle)

Purpose
Accepts a stream-style connection from a client. This function should be issued by the server in response to a *CR_RegisterClient* function that has been issued by the client. A communication channel must have been established on both the server and the client beforehand.

Parameter
nReceivingHandle
The handle onto the communication channel.

Return values
A secondary handle if the communication was accepted successfully. If there was a communication problem, then *COMM_ERROR* is returned, else if a communication channel has not been previously registered, then *LINK_ERROR* is returned.

Error codes
CESTREAMONLY
This function can only be used with stream communication channels.
CECANTACCEPT
The server could not accept correctly with a client.
LECANTADDHANDLE
The secondary handle to be associated with this communication channel could not be created.
LECANTGETHANDLE
The primary handle associated with this communication channel could not be located.
LECANTADDINFO
The registration information associated with this communication channel could not be added.

Notes
This function should only be used on stream-style connections. A server can only accept on one connection at a time for each primary handle.

See also
CR_Unaccept

5.5. INT CR_Send (int nHandle, LPHSTR lphszData, INT IBytesToSend, int nTimeout)

Purpose
Sends data to a communication channel.

Parameters
nHandle
The handle onto the communication channel.
lphszData
The data to send to the communication channel.
IBytesToSend
The amount of data to send to the communication channel.
nTimeout
The amount of time (in seconds) before the connection will timeout.

Return values
The number of bytes sent if the data was written successfully. If there was a communication problem, then *COMM_ERROR* is returned, else if the communication channel has not been previously registered, then *LINK_ERROR* is returned.

Error codes
CECANTWRITEDATA
The data could not be send to the communication channel.
CECANTREADDATA
The protocol between the client and the server could not be established.
CETIMEOUT
The timeout period has expired.
CEDATATOOBIG
The data to be sent is too large for the communication channel (for datagram and broadcast communication channels only).
LECANTALLOCMEM
Memory allocation failed.
LECANTGETHANDLE
The primary or secondary handle associated with this communication channel could not be located.

Notes
A timeout is not considered to be an error, even though *CR_GetLastError* will return *CETIMEOUT*.

See also
CR_Receive; *CR_ReceiveN*

5.6. INT CR_Receive (int nHandle, LPHSTR lphszData, INT IBytesToRead)

Purpose
Reads data from a communication channel.

Parameters
nHandle
The handle onto the communication channel.
lphszData
The resultant data that has been read from the communication channel.
IBytesToRead
The amount of data to read from the communication channel.

Return values
The number of bytes read if the data was read successfully. If there was a communication problem, then *COMM_ERROR* is returned, else if the communication channel has not been previously registered, then *LINK_ERROR* is returned.

Error codes
CECANTREADDATA
The data could not be read from the communication channel.
CECANTWRITEDATA
The protocol between the client and the server could not be established.
LECANTALLOCMEM
Memory allocation failed.
LECANTGETHANDLE
The primary or secondary handle associated with this communication channel could not be located.
LECANTADDINFO
The registration information associated with this communication channel could not be added or updated.

Notes
If there is no data waiting at the communication channel, then this function will block until it arrives. However, it will **not** block until all of the requested data arrives (see *CR_ReceiveN* for a fully blocking

receive function). Additionally, this function can only send stream data of a maximum size of 32767 bytes (use *CR_ReceiveN* to send data that is over this limit).

See also
CR_Send; *CR_ReceiveN*

5.7. INT CR_ReceiveN (int nHandle, LPHSTR lphszData, INT IBytesToRead, int nTimeout)

Purpose
Attempts to read data from a communication channel until all of the required amount of bytes have been read or the timeout has been reached.

Parameters
nHandle
The handle onto the communication channel.
lphszData
The resultant data that has been read from the communication channel.
IBytesToRead
The amount of data to read from the communication channel.
nTimeout
The amount of time (in seconds) before the connection will timeout.

Return values
The number of bytes read if the data was read successfully. If there was a communication problem, then *COMM_ERROR* is returned, else if the communication channel has not been previously registered, then *LINK_ERROR* is returned.

Error codes
CECANTREADDATA
The data could not be read from the communication channel.
CECANTWRITEDATA
The protocol between the client and the server could not be established.
CETIMEOUT
The timeout period has expired.
LECANTALLOCMEM
Memory allocation failed.
LECANTGETHANDLE
The primary or secondary handle associated with this communication channel could not be located.
LECANTADDINFO
The registration information associated with this communication channel could not be added or updated.

Notes
If there is no data waiting at the communication channel, then this function will block until it arrives. A timeout is not considered to be an error, even though *CR_GetLastError* will return *CETIMEOUT*.

See also
CR_Send; *CR_Receive*

5.8. int CR_GetInfo (int nHandle, LPINT lpiPort, LPSTR lpszIPAddress, int nInfoType)

Purpose
Interrogates either the local or remote address structure of a registered communication channel, yielding the port number and Internet address.

Parameters
nHandle
The handle onto the communication channel.
lpiPort

The resultant port number of the remote machine.
lpszIPAddress
The resultant Internet address of the remote machine.
nInfoType
The type of information to return. If this is *CR_LOCAL*, then the details of the local host are returned, else if this is *CR_REMOTE*, then the details of the currently connected remote host are returned.

Return values
TRUE if the remote address structure was interrogated successfully. If there was a problem retreiving the local or remote machine information, then *COMM_ERROR* is returned, else if there is no complete remote address structure associated with the communication channel, then *LINK_ERROR* is returned.

Error codes
LECANTQUERYHNDL
The communication channel could not be interrogated to determine additional registration information.
LECANTGETINFO
The registration information associated with this communication channel could not be located.

Notes
The remote address structure for a stream-style connection is completed when a *CR_Accept* function has been issued (and the connection is made). For a datagram-style connection, the remote address structure is updated each time a *CR_Receive* is issued (since many different clients can connect to one server). This means that until the first client sends data, the remote address structure for a datagram-style connection will be incomplete.

5.9. int CR_GetIPAddress (LPSTR lpszHostName, LPSTR lpszIPAddress)

Purpose
Looks up the Internet address of a machine within the local name server or local hosts file.

Parameters
lpszHostName
The host name of the machine to look-up.
lpszIPAddress
The resultant Internet address of the specified machine

Return values
TRUE if the Internet address of the machine was looked-up successfully. If there was a problem obtaining the Internet address, then *LINK_ERROR* is returned.

Error code
LECANTQUERYHOST
The local host could not be queried for relevant information.

5.10. int CR_GetLastError (void)

Purpose
Returns an error code that is associated with the last error that occurred.

Return value
CEOKAY if no error occurred, else the error code associated with the last error.

Notes
Two types of errors can be returned by the Communication Routine functions; communication errors (*COMM_ERROR*) and link errors (*LINK_ERROR*). A communication error indicates that the communication subsystem has failed and a link error means that there was a problem setting up the local registration information within the link handler. Error codes preceded with *CE* detail a communication error and error codes preceded with *LE* represent a link error.

5.11. int CR_Unaccept (int nSecondaryHandle)

Purpose
Disconnects the server from a stream-style connection with the client. The server is now free to accept another stream-style connection from another client or to unregister its primary handle.

Parameter
nSecondaryHandle
The secondary handle to disconnect.

Return values
TRUE if the secondary handle was disconnected successfully. If there was a communication problem, then *COMM_ERROR* is returned, else if the secondary handle does not exist, then *LINK_ERROR* is returned.

Error codes
CESTREAMONLY
This function can only be used with stream communication channels.
CECANTCLOSEHNDL
The communication channel could not be unregistered or closed.
LECANTGETHANDLE
The secondary handle associated with this communication channel could not be located.
LECANTDELHANDLE
The secondary handle associated with this communication channel could not be removed.

Notes
A secondary handle should be unaccepted before its corresponding primary handle is unregistered.

See also
CR_Accept; *CR_Unregister*

5.12. int CR_Unregister (int nPrimaryHandle)

Purpose
Unregisters a communication channel and closes the connection with the attached remote machine.

Parameter
nPrimaryHandle
The primary handle to disconnect.

Return values
TRUE if the primary handle was disconnected successfully. If there was a communication problem, then *COMM_ERROR* is returned, else if the primary handle does not exist, then *LINK_ERROR* is returned.

Error codes
CECANTCLOSEHNDL
The communication channel could not be unregistered or closed.
LECANTGETHANDLE
The primary handle associated with this communication channel could not be located.
LECANTDELHANDLE
The primary handle associated with this communication channel could not be removed.

Notes
A secondary handle should be unaccepted before its corresponding primary handle is unregistered.

See also
CR_Unaccept

5.13. int CR_UnregisterAll (void)

Purpose
Unregisters all communication channels and closes the connections with all attached remote machines.

Return values

TRUE if all of the connections were unregistered and closed successfully. If any remote machine could not be unregistered, then *LINK_ERROR* is returned.

Error codes
LECANTDELHANDLE
A primary or secondary handle could not be removed.

Notes
All secondary handles should be unaccepted before this function is called. *CR_Unregister* should be used to unregister individual connections.

See also
CR_Unaccept; *CR_Unregister*