
Building Networked Applications with the LabWindows™/CVI™ TCP Support Library

Introduction

This document describes how to use the LabWindows/CVI TCP Support Library to build networked applications.

TCP/IP

The Transmission Control Protocol/Internet Protocol (TCP/IP) suite is a set of protocols that govern how data is transferred between networked computers. TCP/IP is the primary protocol of the Internet, the Web, and many private networks and local area networks. Networking components rely on TCP transport for many activities, including the following operations:

- Internet access using HTTP
- Access to remote file servers and printers
- Distributed Component Object Model (DCOM) services

TCP-Based Network Applications

Typically, network communication using TCP involves a client-server architecture. A client or server can be any device on the network identified by a logical IP address. Before creating a TCP network connection, the server application must register on the network. Registration establishes the port through which client applications can access the server. After registration, the TCP server application listens on the network for incoming client requests. To issue a request to a server, the client must know the name or network address of the host machine on which the server application is running and the server port number. When a server receives a request, it processes the request and replies to the client. Figure 1 shows the interaction between a TCP server and TCP client application. Because TCP is connection-based, the server and client must connect with each other before they can exchange data.

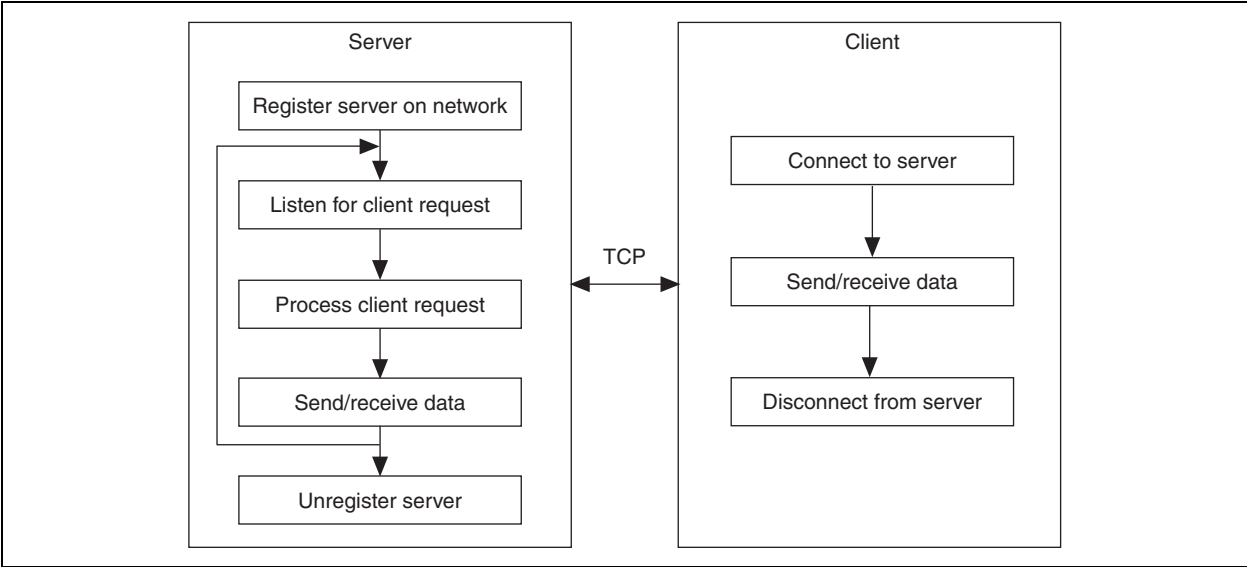


Figure 1. TCP Communication

Components and Dependencies in a LabWindows/CVI TCP Application

Figure 2 shows important components and dependencies in a LabWindows/CVI TCP application.

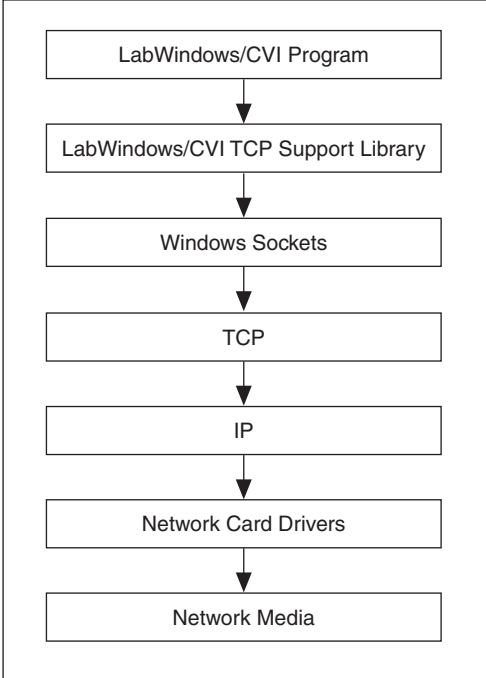


Figure 2. TCP Components and Dependencies

Programs you write using the LabWindows/CVI TCP functions depend on the LabWindows/CVI TCP Support Library, which is part of the LabWindows/CVI Run-Time Engine. The LabWindows/CVI TCP Support Library uses the Windows Sockets (Winsock) API. Winsock provides direct access to services in the transport layer using the TCP/IP protocol. TCP establishes a connection for data transmission, and IP defines the method for sending data packets.

Creating TCP Applications Using the LabWindows/CVI TCP Support Library

The LabWindows/CVI TCP Support Library provides easy-to-use functions to create TCP server and client applications. Callback functions provide the mechanism for receiving notification of connection initiation, connection termination, and data availability. The LabWindows/CVI TCP Support Library calls the server and client program callback functions when the following TCP events occur:

- `TCP_CONNECT` – Occurs when a new client requests a connection. This event can occur only in a server application.
- `TCP_DATAREADY` – Occurs when data is available to be read.
- `TCP_DISCONNECT` – Occurs when a client or server terminates the connection.

Figure 3 shows how to use the LabWindows/CVI TCP Support functions to create a TCP server and client.

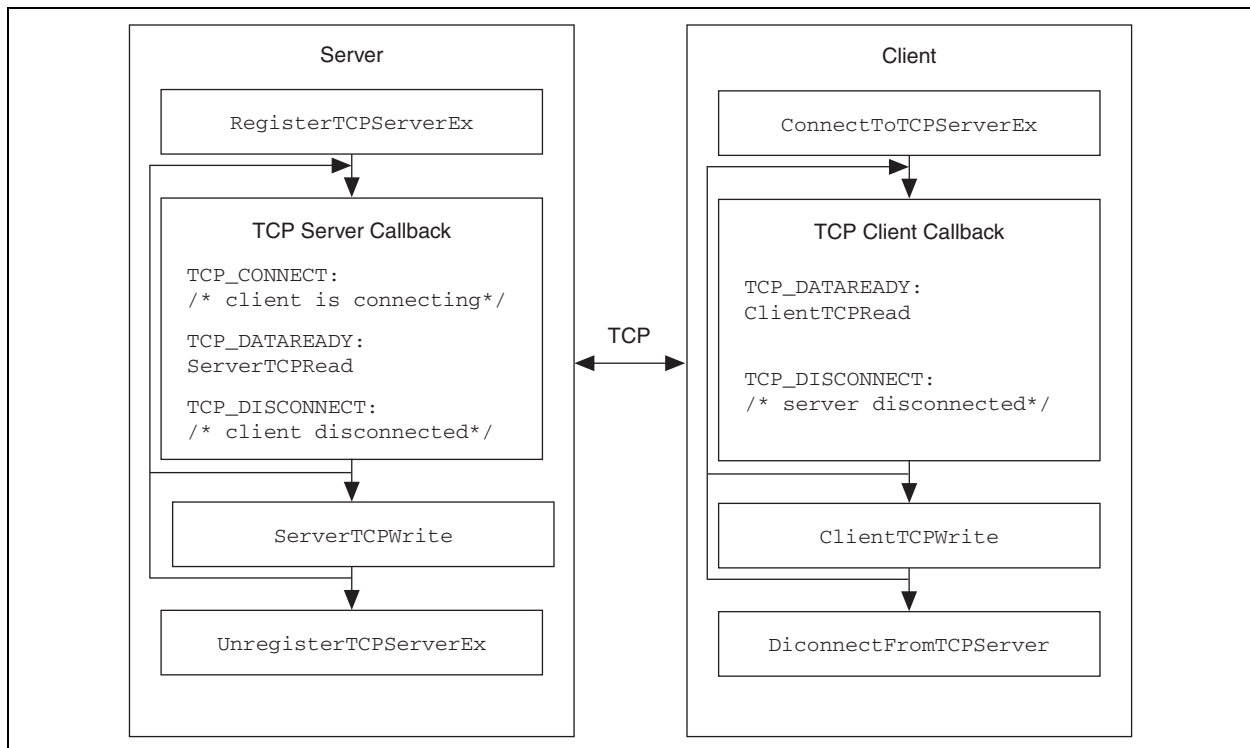


Figure 3. LabWindows/CVI TCP Support Functions

The communication process begins when an application registers itself as a valid server. The client then connects to the server using the port number specified in the server registration function. When the client connects to the server, the server application receives the `TCP_CONNECT` event. Once the connection is established, both the server and the client applications can receive `TCP_DATAREADY` events, which indicate to the applications to call their respective read functions. When the server disconnects from the client, the client application receives the `TCP_DISCONNECT` event.

Similarly, when the client disconnects from the server, the server application receives the `TCP_DISCONNECT` event. The client and server applications also can receive the `TCP_DISCONNECT` event if the connection terminates because of an error.

You can use TCP only for one-to-one communication. You can connect a server or client application to several other client or server applications concurrently; however, only one server and one client can exchange data in a single communication session. In the LabWindows/CVI TCP Support Library, a *conversation handle* represents an individual communication session. In a client application, the `ConnectToTCPServer` and `ConnectToTCPServerEx` functions return conversation handles when the client connects to the server application. In a server application, the LabWindows/CVI TCP Support Library passes the conversation handle to the TCP callback function when the function receives a `TCP_CONNECT` event, which occurs when a new client connects to the server. Note that 0 is a valid conversation handle.

TCP Data Transfer

Message-Based Communication

TCP is a stream-oriented, binary data transfer protocol. TCP does not have any special built-in capabilities for data types such as floating-point numbers, structures, arrays, and so on. The streaming nature of the protocol results in data being fragmented and coalesced among packets. Therefore, if you send 100 8-byte double-precision numbers in one write call from a client, the server might not get all the 100 8-byte double-precision numbers in one read call. Furthermore, because TCP is not aware of data types, a read call might not even receive the data in 8-byte boundaries.

You must design the reading code to handle such cases. The data length is not an issue if both the reading code and the writing code know the size of each data transfer because the reading code can keep reading the data until all the bytes are received. However, writers sometimes must send data of varying lengths. In these cases, you can send the data in a message with size information encoded in the header of the message. For example, consider that each message has a 1-byte header that represents the number of eight-byte double-precision numbers following it in the message. The following code reads such a message correctly in a server program's TCP callback function:

```
int CVICALLBACK TCPCallback (unsigned handle, int event, int error, void *callbackData)
{
    int bytesToRead, bytesRead, dataSize;
    unsigned char numPoints;
    char*data = NULL;

    switch (event)
    {
        case TCP_CONNECT:
            /* Client connected */
            break;
        case TCP_DATAREADY:
            /* Read the number of points in this data */
            ServerTCPRead (handle, &numPoints, sizeof (numPoints), 0);

            /* Read the data. Note that the data could be empty. */
            if (numPoints > 0)
            {
                dataSize = bytesToRead = sizeof (double) * numPoints;
                data = malloc (dataSize);
                if (data == NULL)
                {
                    MessagePopup ("Error", "Out of memory!");
                    goto Done;
                }
            }
        }
    }
}
```

```

* NOTE:
* It is possible that not all the data has
* been received. Therefore, we want to ignore timeout
* errors and continue reading until we get
* all the data.
*/
DisableBreakOnLibraryErrors ();
while (bytesToRead > 0)
{
    bytesRead = ServerTCPRead (handle, &data[dataSize -
    bytesToRead], bytesToRead, 0);
    if (bytesRead > 0)
        bytesToRead -= bytesRead;
    else if (bytesRead != -kTCP_TimeOutErr)
        goto Done;
}
EnableBreakOnLibraryErrors ();

/* Done reading. Do something with the data. */
YGraphPopup ("Data", (double*)data, (int)numPoints,
VAL_DOUBLE);

    break;
case TCP_DISCONNECT:
    /* Client disconnected */
    break;
}

Done:
    free (data);
    return 0;
}

```

Because TCP is a binary data transfer protocol, all data is assumed to be in bytes. Therefore, you must consider issues such as Endianess (machine byte order) when you send and receive data.

Refer to the LabWindows/CVI `samples\tcp\message.cws` sample program for more information about designing message-based TCP servers and clients.

Streaming Nature of TCP

In TCP, because data is sent in a byte stream, it is not possible to know how many bytes are present in the data stream. Therefore, you cannot query the LabWindows/CVI TCP Support Library to get the amount of data available. When you call `ServerTCPRead` or `ClientTCPRead`, the library returns the number of bytes read into the data buffer. If the number of bytes read is the same as the data buffer size, then more data still might be present in the data stream. You can read the remaining data immediately by calling the read functions again or wait until the TCP callback is called again with the `TCP_DATAREADY` event. The LabWindows/CVI TCP Support Library calls the TCP callback with the `TCP_DATAREADY` event whenever events are processed and there is data in the TCP stream. Refer to the LabWindows/CVI `samples\tcp\message.cws` sample program for code that shows how to read all the data in the TCP stream whenever the TCP callback receives a `TCP_DATAREADY` event. The `ServerTCPWrite` and `ClientTCPWrite` functions return the number of bytes written to the TCP stream, which might be less than the number of bytes in your data buffer. You must check for this condition and make sure all the bytes in your data buffer are written out successfully. The following code, based on code in the `samples\tcp\messagewriter.c` sample file, demonstrates how to check for this condition.

```

int    numPoints, dataSize;
int    *data = NULL;
int    index;
int    bytesToWrite, bytesWritten;

/* Get the number of points to write and allocate memory */
GetCtrlVal (gPanel, PANEL_NUM_POINTS, &numPoints);
dataSize = (numPoints + 1) * sizeof (int);
data = malloc (dataSize);
if (data == NULL)
    {
        MessagePopup ("Error", "Could not allocate memory for data!");
        goto Done;
    }

/* Set the number of points and generate the data */
data[0] = numPoints;
for (index = 1; index <= numPoints; ++index)
    data[index] = rand() % 100;

/* Send all the data */
bytesToWrite = dataSize;
while (bytesToWrite > 0)
    {
        bytesWritten = ServerTCPWrite (gConversationHandle, &data[dataSize -
        bytesToWrite], bytesToWrite, 0);
        if (bytesWritten < 0)
            goto Done; /* TCP error occurred */
        bytesToWrite -= bytesWritten;
    }

/* Display the data */
DeleteGraphPlot (gPanel, PANEL_GRAPH, -1, 0);
PlotY (gPanel, PANEL_GRAPH, &data[1], numPoints, VAL_INTEGER, VAL_CONNECTED_POINTS,
VAL_EMPTY_SQUARE, VAL_SOLID, 1, VAL_YELLOW);

Done:
free (data);

```

Note that a single-read function can receive multiple data messages or parts of messages. Therefore, the application that sends the data can indicate to the application that receives the data where the message ends by specifying the message size or passing a special character to indicate the end of a message.

Explicitly Specifying a Client Port

Unlike TCP servers, it is not necessary for TCP clients to explicitly specify or register a port for the connection. However, because of the presence of network firewalls and so on, you might want to specify the local host port the client program uses to establish a connection to the TCP server. To specify the client port, you must use the `ConnectToTCPServerEx` function, which includes the **clientPort** parameter. This parameter is different from the **serverPort** parameter, which you must specify correctly no matter which function you use. If you specify the client port explicitly, you must make sure that other TCP programs are not using that client port. National Instruments recommends that you not explicitly specify the client port unless it is necessary to use a specific client port. Use `TCP_ANY_LOCAL_PORT` if you do not need to specify the client port explicitly.

Multithreading and Serving Multiple Clients

The LabWindows/CVI TCP Support Library is multithread safe. In many applications, it makes sense to separate the TCP tasks from other tasks, such as user-interface management, by running the different tasks in different threads. You can design TCP servers that serve multiple clients simultaneously to process each client in a separate thread. This prevents one client from overloading the server and causing the server to exclude other clients. If one client issues a time-consuming task request, the server can continue to respond to other clients in the other threads. When designing multithreaded TCP programs, note that the TCP callback functions are called in the same thread in which `RegisterTCPServer (Ex)` or `ConnectToTCPServer (Ex)` are called. Therefore, that thread must process TCP events by calling `RunUserInterface`, `ProcessSystemEvents`, or `ProcessTCPEvents`. Refer to the LabWindows/CVI `samples\tcp\multithreading.cws` sample program for details about how to design multithreaded TCP applications. The `multiclientserver.cws` TCP sample program is a simple TCP server that accepts multiple clients and services each client's requests in a different thread.

Writing TCP Applications on Multihomed Hosts

In some cases, your host machine might have multiple network interfaces, including network cards, adapters, and so on, that might be connected to the same or multiple networks. In these cases, the host machine is said to be *multihomed*. If you are writing a TCP server program on a multihomed host, then National Instruments recommends that you call `RegisterTCPServerEx` instead of `RegisterTCPServer`. Using `RegisterTCPServerEx`, you specify the IP address of the network interface to use on the local host for the server, whereas `RegisterTCPServer` uses the network interface configured as the default interface in your system. You can use the Network Connections applet of the Windows Control Panel to change the default interface. Call the LabWindows/CVI TCP Support Library `GetAllTCPHostAddresses` function to get the IP addresses of all the network interfaces on your local machine.

If you are writing a TCP client program on a multihomed host, you cannot specify the network interface to use for the connection in the LabWindows/CVI TCP Support Library. The network routing tables configured on your host machine restrict this choice. However, you can use the low-level Windows command-line utility program called `route` to change the routing tables and control the network interface to use for connecting to a particular server.

Using the TCP Support Library in the LabVIEW Real-Time Module

You can use LabWindows/CVI to build DLLs for LabVIEW Real-Time targets. The LabWindows/CVI TCP Support Library is fully compatible with LabVIEW Real-Time targets. However, the LabWindows/CVI User Interface Library is not compatible with these targets. Therefore, you cannot call `RunUserInterface` or `ProcessSystemEvents` to process messages and TCP events. Instead, you can call the LabWindows/CVI TCP Support Library `ProcessTCPEvents` function to process TCP events on LabVIEW Real-Time targets. You must call this function in a polling loop to process TCP events and ensure that your TCP callback function is called appropriately on LabVIEW Real-Time targets. Refer to the LabWindows/CVI `samples\tcp\rtServer.cws` sample for details about writing a TCP server to run on the LabVIEW Real-Time platform.

Error Handling

TCP provides end-to-end error detection and correction and reliable data delivery. Specifically, TCP notifies the sender of packet delivery, guarantees that packets are delivered in the same order in which they were sent, retransmits lost packets, and ensures that data packets are not duplicated. TCP uses a checksum on both the headers and data of each segment to help detect network corruption.

The LabWindows/CVI TCP Support Library provides important error and status information in the return value of the functions. Zero and positive return values indicate that the operation succeeded. The read and write functions return the number of bytes transferred on successful completion. Negative return values indicate that an error occurred. When an error occurs, you can call the `GetTCPErrString` function to get a description of a LabWindows/CVI TCP error code. Because errors can occur in the underlying Winsock or other operating system components, you can obtain additional error information by calling `GetTCPSystemErrString` immediately after calling the TCP Support Library function that failed.

Low-Level Socket Programming

You might want to set some options or perform some operations on the underlying connection sockets that the LabWindows/CVI TCP Support Library does not offer. You can call the LabWindows/CVI TCP Support Library `GetHostTCPsocketHandle` function to get the socket handle of a connection and then use the socket handle to call low-level socket functions. You should be familiar with system socket programming if you call low-level socket functions. System socket programming is beyond the scope of this document and is not described here. Refer to the MSDN Windows SDK documentation for more information about TCP and system socket programming.

