Client-Server Applications in LabVIEW

Gavin Burnell 28th February 2000

Introduction

National Instruments LabVIEW is a very flexible programming environment for instrumentation development. In version 5.x LabVIEW has gained functionality to facilitate inter-process communication. In this document, I illustrate how this can be used to build a client-server system where the server process runs asynchronously.

Interprocess Communication

Version 5.x of LabVIEW introduces a number of inter-process communication methods, which I discuss below. All of them, however, follow a similar pattern of use; creation of a refnum, writing or generation of an event, reading or waiting for event with a timeout, and deletion of refnum and freeing of resources.

In the context we are discussing here, a process is an independently running LabVIEW routine, however, all such processes are launched from the same LabVIEW instance (either the development environment, or a runtime engine).

Notifiers

A notifier is a mechanism for passing transitory data from one process to another. The data contained in the notifier may be overwritten before the reader reads it. If there is more than one process waiting for a given notifier then they all receive

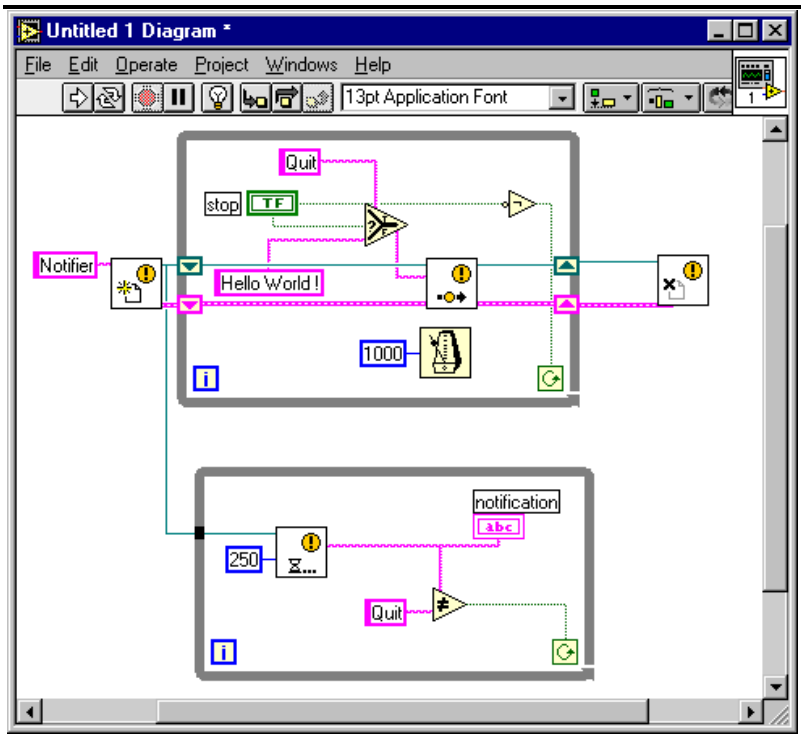


Figure 1. A Simple Example of using a notifier to communicate between two parallel loops, showing creation, sending and waiting on a notification, and finally destroying the notifier.

the data sent. This process is somewhat analogous to sending data on a UDP

broadcast, in that neither sender nor receiver should assume that all data that was sent was received. A notifier is well suited for transmitting informational messages about program state between processes.

Queues

Unlike notifiers, queues guarantee transmission of data from the sender to a receiver. Also, unlike notifiers, it is not possible to transmit information from one sender to multiple readers. The queue is somewhat analogous to a TCP based protocol, in that information is reliably transferred from one sender to one reader, however, further control logic would be required to hold a two-way data exchange over a queue to prevent a process reading its own sent data. For our client-server application, the queue is the basis of the communication.

Semaphores and Rendezvous

Semaphores and Rendezvous are used to control program flow. The semaphore is best suited to ensuring that only one process is carrying out a given task at any one time (for example accessing a piece of hardware). A rendezvous, on the other hand, is most appropriate for synchronising separate processes (for example to synchronise a measurement with control of a piece of apparatus).

A Simple Read-Only Server

The simplest form of server would be a process that only listened for incoming messages and then did whatever was appropriate based on the input. Such a

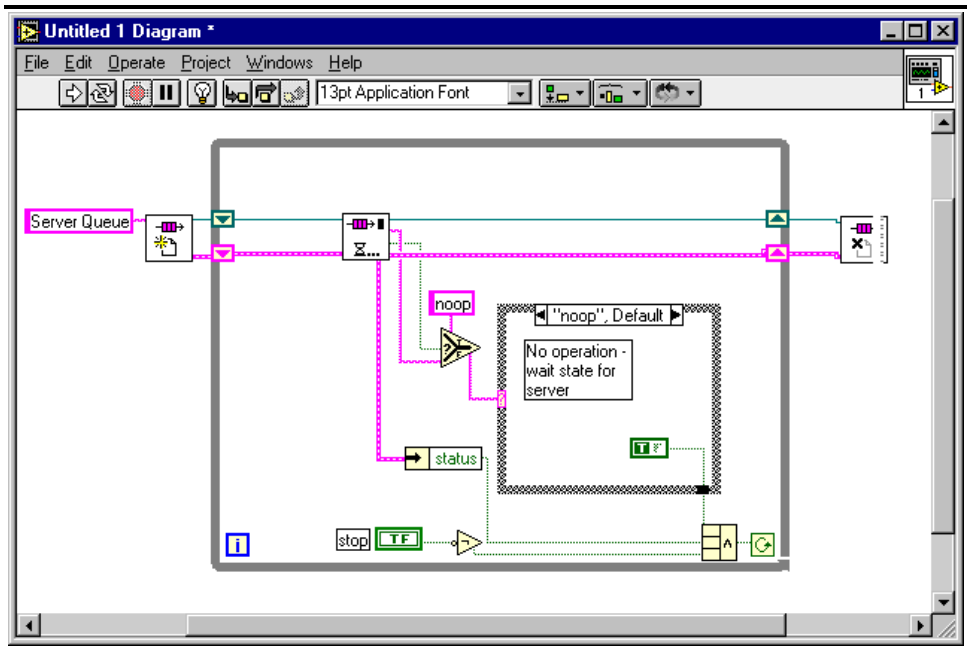


Figure 2. A simple server which reads from a queue and executes commands

server would need only to create a suitable queue, and then attempt to read from the queue whilst running a while-loop. Adding a timeout to the read would allow the loop to cycle on a regular basis to poll a manual stop button.

Clients would communicate with the server by sending simple string messages via the queue (which would have a well defined name).

The obvious disadvantage with this system is that the communication process is only one-way. Even if the process does not inherently require to return data, the client has no way of knowing whether the server process is running and has correctly processed the data. For this reason, a two way communication process is required.

Two Way Communication – A Read-Write Server

As we discussed above, sending two way data via a queue, whilst possible, does require additional flow-control logic. A more elegant method is to use two queues, one for up-stream and one for down-stream communication. It might seem, at first sight, that we could make do with two queues per server, so that the server would receive on one queue and return the results on the other. The problem with this is that is that it fails when more than one client is active. If one client

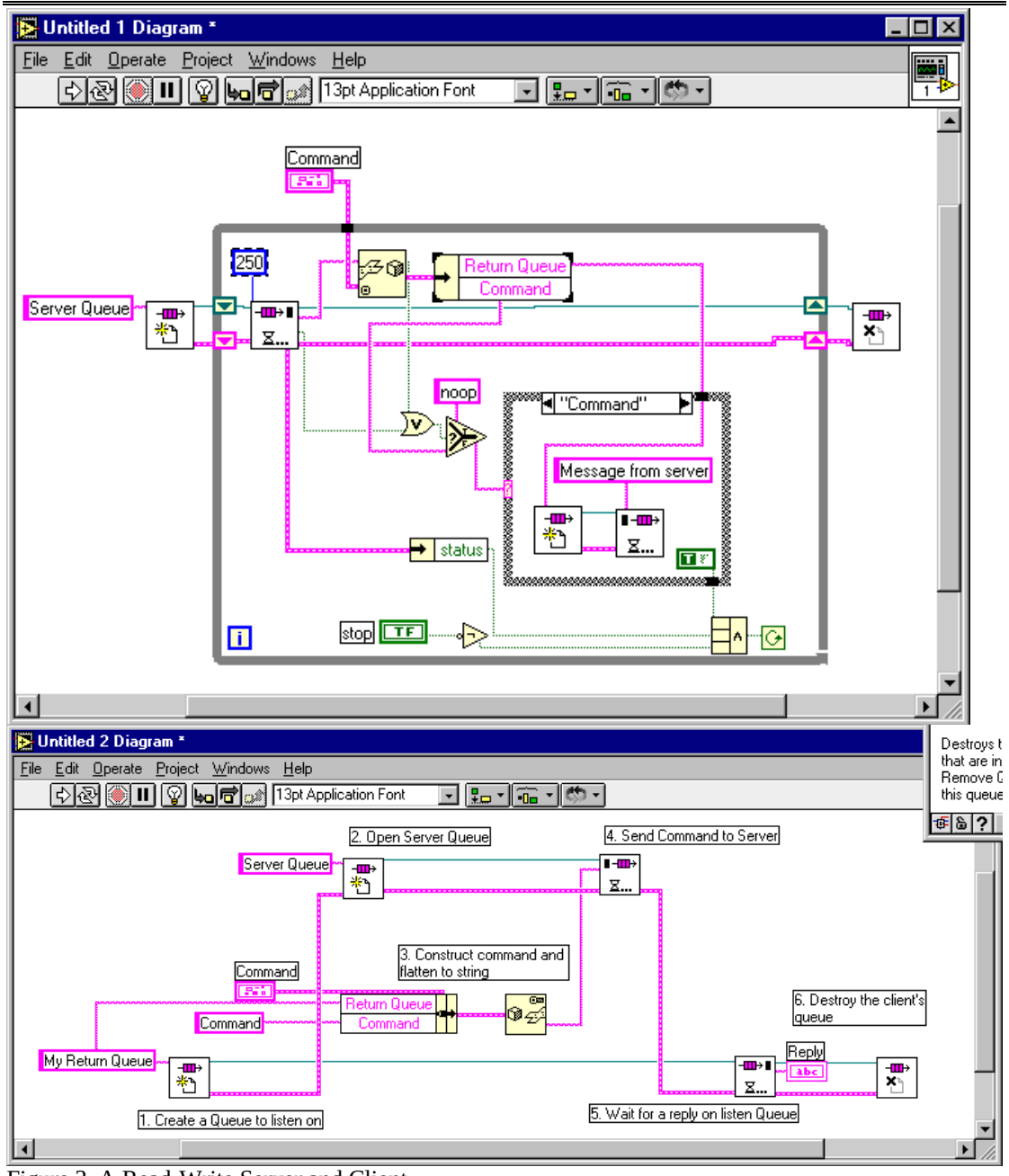


Figure 3. A Read-Write Server and Client.

sends a request to the server, and then starts to wait on the down-stream queue, but before the server has finished processing the request, another client sends a request and starts waiting for its reply, then the two clients have entered a race to receive the answer from the server to client 1. In principle, we should expect that 50% of the time, client 2 will get client 1's data.

To circumvent this, we could tag the replies with an identification to allow clients to recognise their data, and to re-insert other data onto the queue. Such a solution, however, doesn't solve the underlying problem, and is likely to increase the entries being added and removed from the queue. Rather, we should have a separate queue for each client, so that the client only sees data intended for it.

Our process would now be; client creates a listening queue, sends command on server's queue including a reference to its listening queue. The server receives the command from its queue and writes its response to the client's queue. The client listens on its queue for the server. This is illustrated in Figure 3. There is one twist to this scheme. In order to send both a command and a return queue to the server, a cluster structure has been used. However, the queues only work with string data, so we have to flatten the cluster to a string before sending to the server and unflatten on receipt at the server.

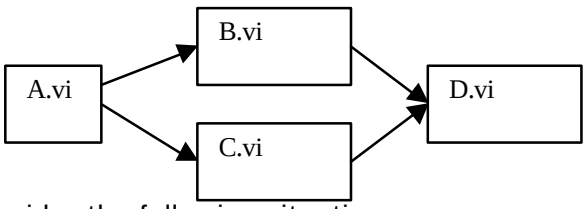
This scheme is perfectly adequate if the number of client calls is small. If the client calls the server many times, then the overhead of creating and destroying the queues gets excessive and memory consumption increases. The solution is to create a pool of queues which clients may used, and then use another queue to hold the names of theses queues. Clients then remove a free client queue-name from this queue of queues, use it to receive a reply from the server, and then return the queue name to the queue of queues when finished. This will limit the number of simultaneous client calls to the number of queues in the pool, however extra code can be added into the server or client to maintain the number of free queues in the pool within in certain constraints.

A Real World Example

As a real example of such a server in use, I have chosen the vi-reference server that is used in many of the programs that I have developed for the Device Materials Group. This program was written to assist the process of dynamic (i.e. runtime) calls to routines. LabVIEW 5.x has the ability to load and call a routine at run-time if the routine's name and path is known. This is accomplished using the vi-server functionality.

Briefly, a reference to the desired routine is opened, a process which loads the routine into memory. By the use of property and method 'nodes' the routine's front panel controls can be maniplulated, its front panel opened or closed, and the routine run. When the last open reference to the routine is closed, the routine will unload.

There are two problems with this scheme that the vi-reference server was designed to overcome. Firstly, one requires either to know the exact path to the routine, or to be able to search the disk on each call to a routine to determine its



path. Secondly, consider the following situation:

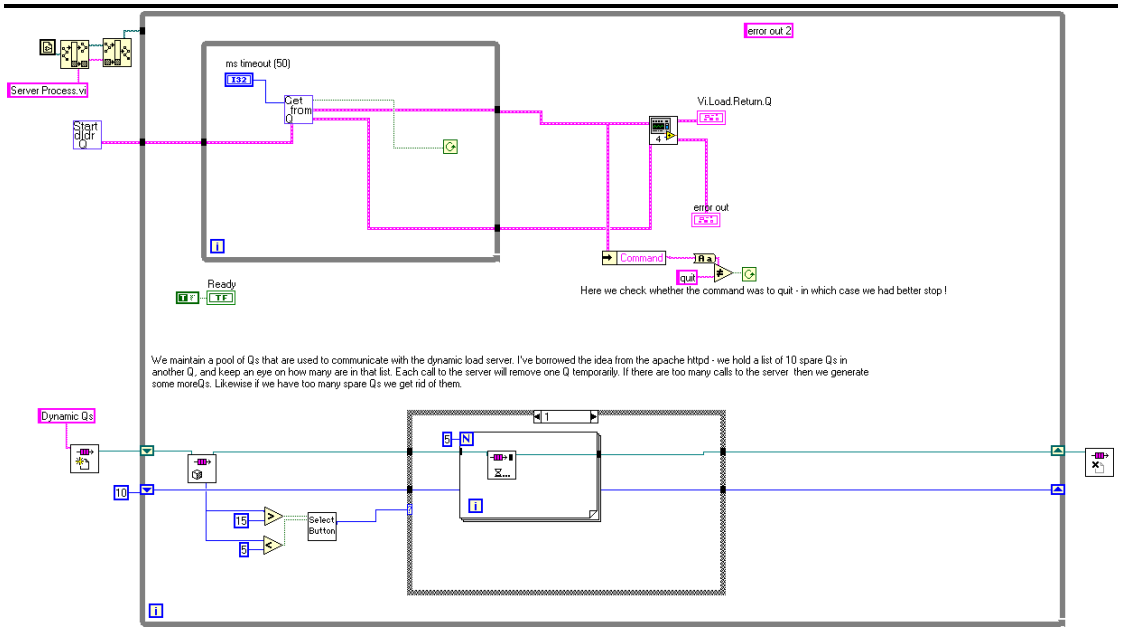


Figure 4. The vi-reference server

A.vi dynamically calls B.vi, loading it into memory. B.vi dynamically calls D.vi, loading it into memory in turn. B.vi finishes, releasing D.vi from memory, which then unloads from memory. C.vi is then called and now has to re-load D.vi into memory. If D.vi is sizeable and contains many sub-vis this can take a while. A better system would have all the loading into memory done by A.vi, but A.vi doesn't know in advance which routines are required, and is busy running B.vi when B.vi wants to load D.vi. The solution is for a helper app to do all the loading, which because it runs asynchronously with respect to A.vi doesn't get blocked. Hence the vi-reference server.

The Vi-Reference Server Process

The vi-reference server (shown in Figure 4) follows the scheme for our Read-Write Server pretty closely. The main differences are that the command interpretation and processing and the replying to the client is all delegated to one sub-vi. In addition, the management of the pool of client queues is maintained in the server.

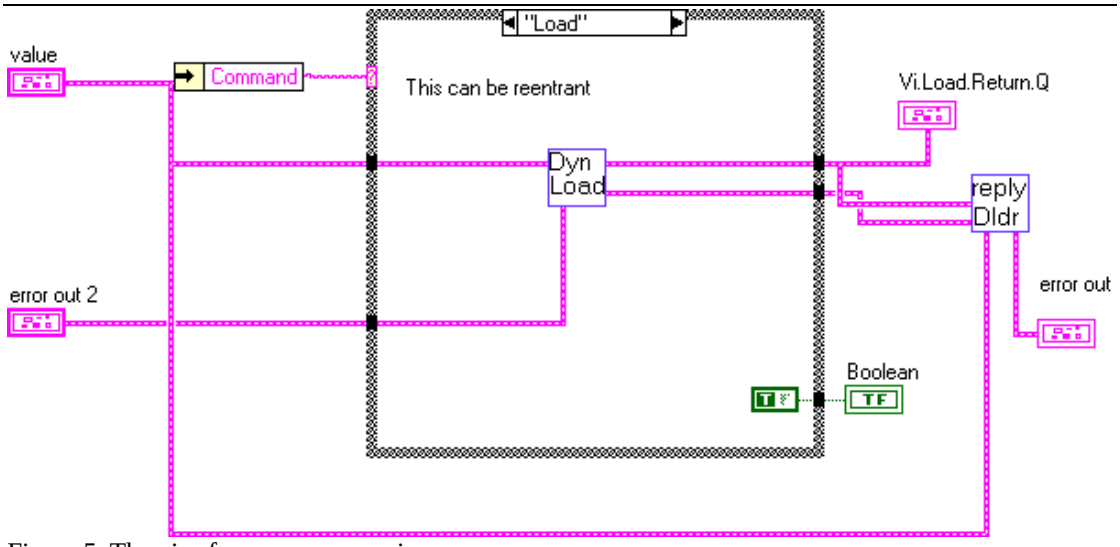


Figure 5. The vi-reference server main process

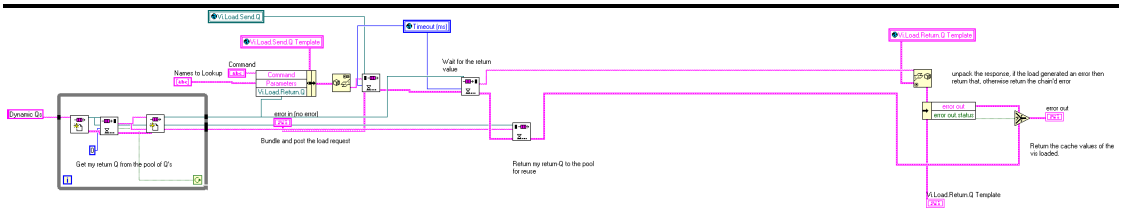


Figure 6. The Client routine

The Vi-Reference Client

The Vi-Reference Client is actually a set of separate routines that each send one command to the server by calling a common sub-routine (shown in figure 6). The client also follows the scheme closely, using the pool of return queues to get a queue to listen on. We make use of some global variables to store static information such as the server queue name, timeout values and templates for the sent and returned data structures. Unlike our simple example, the returned data is a cluster structure as well, to facilitate multiple return parameters.

Further Thoughts

Inter-Machine Client-Servers

In this document we have discussed only client-servers running within one instance of LabVIEW. Inter-machine Client-Server setups are an example of the more general case of inter-LabVIEW client-servers. The approach used here will not work directly across LabVIEW isntances, because the queue refnums are specific to a particular instance.

One alternative approach would be to build a complete TCP/IP based client-server system using a custom protocol. This would however add substantially to the complexity of the code. On Win32 platforms, the Data-Socket vis could be used to reduce the problem to a relatively simple case, in which the queue vis would be replaced with Data-socket vis.

For a truly cross-platform approach, one could use the LabVIEW vi-server functions to invoke the create queue/read queue/write queue on the remote machine. A consistent standard to define on which machine the various queues were created and manipulated would be required. An obvious protocol would be to run the server queue on the server machine and the client queues on the client machine. The clients would then need to send not only the name of their queue but also their machine name and location of the queue vi's.

Persistent Client-Server Exchanges

The setup I described here is orientated towards a single transaction, connectionless design – like for example http 1.0. The client makes a connection to the server, sends a single command and receives a single response. This system can process one client at a time, which is not a problem for the example given. If the server has a prolonged exchange with client, one would want the server to e able to process more than one client at a time.

To adapt the system to work with multiple transactions per connection, one would have to allow the server to create a unique write queue for the client, and to pass this back to the client with the first response. Handling multiple clients takes

rather more effort. If each transaction is short, then the server requires just one main loop than handles all available clients. If the transactions are lengthy, then passing both the up- and down-stream queues to a re-entrant vi that handles the transactions might be appropriate.

Gavin Burnell

28th February 2000