Writing Windows NT Server Applications in MFC Using I/O Completion Ports

Ruediger Asche

September 1996

Abstract

One of the most frequently encountered challenges in writing server applications is the multiple-client bottleneck: How to service multiple clients simultaneously without degrading overall performance. This article discusses two possible solutions to this problem: First, I demonstrate how each client can be associated with one dedicated server thread, and then I discuss input/output (I/O) completion ports, an I/O system extension in Microsoft® Windows NT® that allows asynchronous I/O calls on several communication objects to be managed efficiently. Since I am an aficionado of object-oriented design, I provide C++ classes for everything I discuss. Note that I/O completion ports have been written about before (for example, an excellent discussion is provided in John Vert's article "Writing Scalable Applications for Windows NT" in the MSDN Library) and that there are several code samples available that demonstrate the use of I/O completion ports. I would also like to direct your attention to two samples that demonstrate slightly different uses of I/O completion ports: The UNBUFCPY and SOCKSRV samples in the Platform SDK.

This article focuses on the practical aspects of programming I/O completion ports—namely, how client/server communications can be expressed in terms of I/O completion worker threads and how to code stable servers that use I/O completion ports.

Strategies to Servicing Multiple Clients

A server application is fairly meaningless if it cannot service multiple clients at the same time. In general, you code a server application such that each client is associated with one dedicated communication object such as a named pipe, a socket, or a remote procedure call (RPC) session. The challenge now is this: How do we code the server such that each communication object is as responsive as possible? The two concepts that are crucial to tackling this challenge are asynchronous I/O and multithreading.

In order to make the following discussion understandable, let's assume that you are writing a database server. Very roughly, your server will do what is depicted in Figure 1: Each time a client connects, the server opens a channel through which it communicates with that client and then enters a loop that runs until the client disconnects. The loop repeatedly requests database commands from the client, then executes the commands locally, and finally returns the results from the commands to the client. Note that from the point of view of the server, the loop is nothing but a repetitive sequence of I/O calls and accesses to the database. Note that this way of looking at client/server interactions is really independent of the type of server; whether you code a database server, an information server, a multimedia server, or whatever, each client/server interaction can normally be reduced to this loop of accepting commands and executing them.

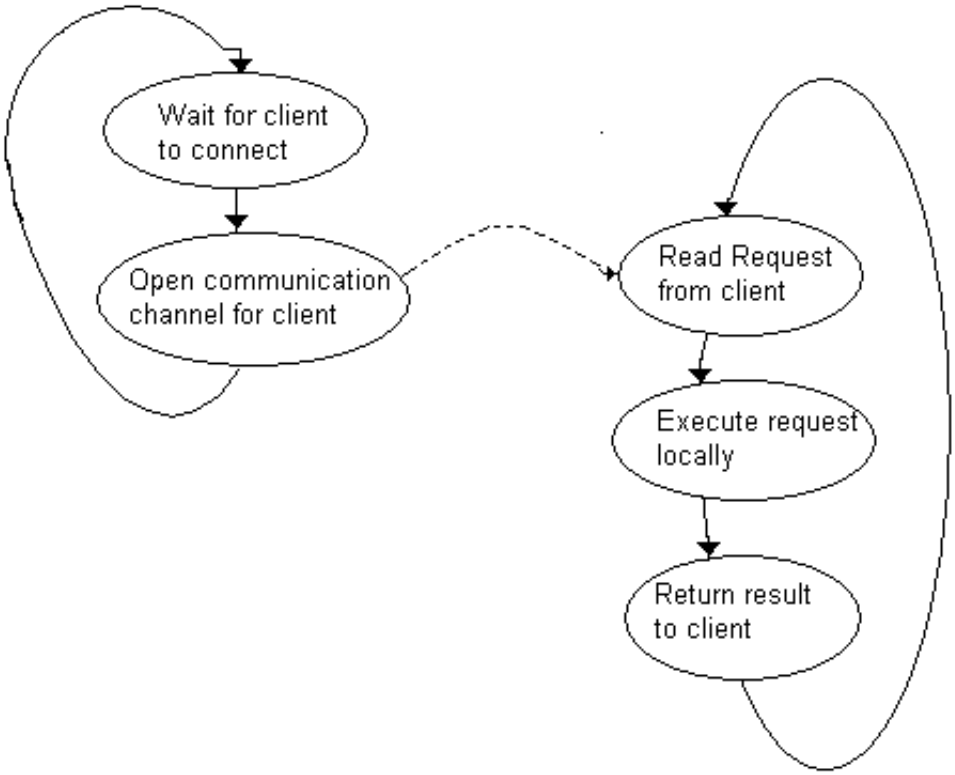

Figure 1. A sample database server application

Back in the days where there was no multithreading, such a server was a weird beast to design because it is not really possible to code a single-threaded server application without introducing some kind of convoy effect: Since the I/O requests of several attached clients must wait for one another, each client will perceive the responsiveness of the server in terms of how the other clients behave.

Oh, yeah, there is also asynchronous I/O, which allows several I/O calls to be pending at the same time, which relieves the convoy effect a little bit. Before we go on, let's have a look at asynchronous I/O quickly because in Microsoft® Win32® there are actually *four* different ways to use it. By definition, an asynchronous I/O call returns immediately, leaving the I/O call itself pending. The question then is this: How do applications obtain the result of the I/O call? Let's look at the four ways of using asynchronous I/O and how each way answers the question:

1. *Using events*. When an application submits a **ReadFile** or **WriteFile** call on an I/O object that was opened with the **FILE_FLAG_OVERLAPPED** flag, and the last parameter to the I/O call points to a valid **OVERLAPPED** structure, the I/O call returns immediately, and the I/O object and (if present) the event object in the **OVERLAPPED** structure gets signaled as soon as I/O completes. Thus, by using an appropriate waiting mechanism, the application can synchronize its I/O. Note that when using this approach, your application can use different threads to submit and process the I/O call.
2. *Using the* **GetOverlappedResult** *function*. Strictly speaking, this strategy should be labeled 1a instead of 2 because it uses the same blocking mechanism as strategy 1. A thread that submits **GetOverlappedResult** with **TRUE** as the last parameter behaves pretty much as if it had called **WaitForSingleObject** on the file object or the event object in the **OVERLAPPED** structure (thus, it's about the same behavior as 1). The **GetOverlappedResult** strategy also comes in a "polling" version, where the function simply checks on the state of the asynchronous I/O and returns immediately. To get this behavior, simply call **GetOverlappedResult** with the last parameter set to **FALSE**.
3. *Using asynchronous procedure calls (or APCs)*. An APC is actually sort of a by-product of the Windows NT® architecture and has been adopted by Windows® 95. When your application uses APCs, it asks the operating system's I/O system to call your application back as soon as the asynchronous I/O call has completed. There are three gotchas to APCs. First of all, the APC is always called in the context of the calling thread. Second, in

order for an APC to be executed, the thread must be suspended in a so-called alertable wait state. Third, in order to be able to use APCs, the calling thread must call **ReadFileEx** or **WriteFileEx** instead of the "normal" **ReadFile** or **WriteFile** functions.

4. *Using I/O completion ports*. This is a new technology that is only available in Windows NT and is the subject of this article.

Well, as far as the convoy effect is concerned in a single-threaded solution, asynchronous I/O only defers the problem: At some time, the results of the asynchronous I/O calls *must* be synchronized with the main thread, such that the convoy effect will show up at the completion of the I/O calls, not the submission. In Win32 terminology, that means that a thread that submits an asynchronous I/O call must at some point either call one of the members of the *xxx*Ex function family (**ReadFileEx**, **WriteFileEx**, or **SleepEx**) to allow an asynchronous procedure callback to execute or call **WaitForSingleObject** on an I/O call-supplied event object. In either case, the calling thread is blocked until the I/O completes, and we are back to the convoy effect.

We could work around this problem by combining asynchronous I/O with multithreading: Let's take variation 1 or 2, because, as we stated before, variation 3 requires the APC to be executed in the context of the same thread as the I/O call. In order to use asynchronous I/O more efficiently using multithreading, we could have one thread submit a number of asynchronous I/O calls (for example, one for each connected client) and then dispatch a number of "worker threads." Each of those threads would be suspended until an I/O call has completed. Typically, each worker thread would serve one pending asynchronous I/O call because every worker thread that services multiple asynchronous I/O calls would suffer from a "local" convoy effect. Note that in terms of blocking behavior and thread servicing, this solution would almost be identical to one that services each I/O call *synchronously* in a separate thread.

A server application written for a multithreaded platform that doesn't have I/O completion ports (for example, the Windows 95 operating system) will typically dispatch one dedicated thread for each client that connects. That dedicated thread will spin in a loop similar to the one shown in Figure 1, repeatedly servicing the I/O between the server and that client until the client disconnects (for a real-life example for such a client-server application suite, check the NPCLIENT and NPSERVER code sample in the Platform SDK).

So far, so good. However, there are a few problems with such a "one-thread-per-client" approach. First of all, threads are system resources that are neither unlimited nor cheap. Thus, if a server application must serve a very large number of clients, the number of system resources claimed by the application can seriously impact the server computer. Second, if the threads are CPU-bound (that is, spend most of their time using up CPU cycles, which can easily happen; for example, in the above database example if the database is large and has complex query and retrieval algorithms), it turns out that unless the server application executes on a true multiprocessor machine, it is actually more expensive to execute the computations in different threads than in a single thread. For a further discussion of this phenomenon, please refer to the article "Win32 Multithreading Performance."

Introducing I/O Completion Ports

Thus, we have a problem: We need multiple threads to efficiently service multiple clients, but we need a way to meaningfully limit the number of threads that execute, regardless of how many clients connect. I/O completion ports (or IOCPs, for short) are the perfect solution to this dilemma. Using IOCPs, a server application can service multiple clients by using multiple threads, but not in a one-thread-per-client fashion. An IOCP is an object that can be associated with a number of I/O objects, such as files, named pipes, or sockets. When a thread requests input via the IOCP, the I/O system blocks the calling thread until *any* pending asynchronous I/O on any of the objects has completed.

An IOCP is not a very sophisticated concept—it is basically nothing but a thread synchronization object, similar to a semaphore. An IOCP, as I said before, can be associated with several I/O objects that support asynchronous I/O. Any thread that has access to the IOCP can ask to be suspended on the IOCP until any pending asynchronous I/O call on one of the I/O objects that is associated with the port has completed. The call that a thread uses to request

suspension is the system call **GetQueuedCompletionStatus**. When this call returns, one pending asynchronous I/O that is associated with the port has completed.

Note that "the call returns" is basically all that happens. There is no paperwork that Windows NT does for you: The operating system simply tells your server application that some I/O has completed. Whenever an I/O object attaches to the port, the server application associates a "key" with that I/O object, and when **GetQueuedCompletionStatus** returns, the key is passed back to the caller. Similar to the parameter passed to a thread function when a thread is created, there is no predefined meaning whatsoever to this "key"—it is simply a unique value that identifies the communication object on which the I/O has completed. It is the server application's responsibility to keep track of what the key parameter means and how the key can be used to determine the result of the I/O operation. Frequently, the key is an index into an array of client control blocks; but it can just as well be a pointer to some kind of data structure or anything else you choose. If your server does not need to distinguish between the I/O objects, the key parameter can be a dummy.

Those of you who are familiar with Win32 synchronization may now argue that the **WaitForMultipleObjects** service can accomplish the same effect that IOCPs can: providing a mechanism to suspend a thread until one of several pending asynchronous I/O requests has completed. In a solution based on **WaitForMultipleObjects**, there would be several pending asynchronous I/O calls, each of which would be associated with one event object. The event objects would then be collected in an array that is passed to **WaitForMultipleObjects**.

As far as the subject matter of this article is concerned—solutions to the multiple client problem —**WaitForMultipleObjects** may indeed provide an alternative to I/O completion ports. However, I/O completion ports are much more flexible and easier to use than **WaitForMultipleObjects**. For example, using **WaitForMultipleObjects** does not allow clients to dynamically attach and detach because the array of event handles passed to **WaitForMultipleObjects** cannot change while the calling thread is suspended, and each of the handles must be valid at any time. Another problem is that **WaitForMultipleObjects** will favor I/O calls on objects whose handles come early in the array, because the array is always traversed from the beginning to the end until a signaled event handle is encountered.

Another advantage of I/O completion ports (which is not too relevant for our discussion, however) is that they allow several outstanding I/O calls on the same I/O object. For example, a file copy operation can be broken up into several chunks, each of which is copied as a separate asynchronous I/O instruction. The UNBUFCPY sample in the Platform SDK is an example for this technique.

Writing Code That Uses I/O Completion Ports

Coding IOCPs can be a bit of a hassle. This is because IOCPs are somewhat counterintuitive. Let's look back at the one-thread-per-client strategy I discussed earlier. If every thread has to keep track of only one dedicated communication with a client, the control flow in the server's thread function is very straightforward: Read a client's command, execute it, and then return the value to the client. The thread function simply mirrors the control flow between the server and the client.

However, in a multithreaded scenario using IOCPs, the control flow of a thread function is less straightforward, because there is no relationship between threads and communications. In other words, a worker thread must prepare to be woken up by any I/O call from any client, decode the client from the IOCP return code, determine where in its control logic the client is (that is, what kind of input or output the client expected at that particular time), and then service the request and eventually dispatch another I/O call to return the result from servicing the request. Look at it this way: When coding dedicated threads, you can focus on the interaction between the client and the server (which is reflected in the thread function, as I mentioned before), whereas in the IOCP solution, you must focus on the worker thread, which does *not* reflect a client/server interaction. Later on, I will show you how you can view a client/server interaction as a automaton that can easily be implemented in a client-specific data structure.

Now that you have a shady idea of what IOCPs are, let's see how they look in practice. Once more (in case I haven't made this clear enough), remember that IOCPs are only available in Windows NT version 3.5 and later.

In order to make the discussion less theoretical, let's first look at the sample application suite, IOCPS, and while I discuss the server application design, we will learn about I/O completion ports as we go along.

The sample suite consists of two Microsoft Foundation Class Library (MFC) projects: the SERVER project and the CLIAPP project. Those of you who are familiar with some of the stuff I wrote for the MSDN Library will meet some old buddies again; the application suite is loosely built around a similar client/server application set that I wrote to demonstrate Windows NT security.

Let's first play around with the application a little bit. The server part (Server.exe) must be executed on a Windows NT machine. The client application (Cliapp.exe) can theoretically run on any machine that executes Win32 applications and can connect to the computer running the server application via a network. However, in order to keep the code sample as lean and mean as can be, I nuked all of the security code from the server, which means that a client on a remote machine will not be able to access the named pipe on the server end. Thus, the best thing to do is to start the server and an arbitrary number of instances of the client application on the same Windows NT machine. If you would like to use the client/server application suite over a network, you can either cut and paste the security code from the NPSERVER sample on the Platform SDK into the **CServerNamedPipe::Open()** member function code, or you can use the security classes I wrote for the MSDN Library to open up the server end of the named pipe to remote clients.

Once the client and server applications are started, you can use the **Connect to Server** menu item from the client's **Remote Access** menu to connect to the server application. (Type the name of the machine into the edit box in the dialog box that appears.) You can then use the menu items from the **Database Access** menu to access a database on the server, that is, insert and remove finite records and view the contents. The server application also has the same database options. In short, the server application maintains the database, and the clients can access the database through the network.

The Sample Service Architecture

The client application I won't talk about at all; if you are reasonably familiar with MFC, you should be able to decipher the client code in no time flat. Thus, let's focus on that part of the server application that is relevant for the client/server interactions. The following C++ classes encapsulate everything we need to know here:

- **CServerNamedPipe** (Npipe.cpp and Npipe.h in the COMMON subdirectory) is the server end of the named pipe class. This class is responsible for the lowest end of the communication between the client and the server, namely, the communication channel. **CServerNamedPipe** is multiply derived from **CServerCommunication** (a class that provides server-specific member functions on communication objects, such as **AwaitCommunicationAttempt**) and **CClientNamedPipe**. I discuss the communication object hierarchy in my series on communication in the MSDN Library. In this context, it is important to mention that the named pipe must be created with the FILE_FLAG_OVERLAPPED flag, indicating asynchronous operation. Without this flag, I/O completion ports do not work. Also note that the client implementations of the **Read** and **Write** member functions differ from their respective implementations on the server side: The clients immediately synchronize their asynchronous I/O using the **GetOverlappedStatus** system call, whereas the server side implementations simply submit the asynchronous I/O calls and then return, leaving it to the I/O completion port to pick up the results, as I will explain when I discuss the **CServerDatabaseProtocol** object in a minute. Two things to notice about the implementation of **CServerNamedPipe** are as follows: First, you will notice that the client side implementation of **Read** and **Write** uses **OVERLAPPED** structures that are kept on the stack. This only works because **Read** and **Write** do not return before **GetOverlappedStatus** has synchronized the asynchronous I/O with the calling stack. The server implementation must use **OVERLAPPED** structures that are not kept on the stack, because the **Read** and **Write** member functions that use them return before the I/O has completed. Second, you will notice something strange in the **AwaitCommunicationAttempt** member function on the

server side:

```
m_ol1.hEvent = ((HANDLE)((DWORD)m_hConnectEvent|0x1));
```

The reason why we manipulate the handle this way is only because of the control flow in the sample server application: The I/O completion port that is associated with the named pipe is established *before* a client connects to the pipe, and that means that every asynchronous operation on the pipe, including a **ConnectNamedPipe** call (which is how **AwaitCommunicationAttempt** is implemented), will end up unblocking a thread. The way I code the I/O completion ports makes it undesirable to unblock a thread upon **ConnectNamedPipe** (in other words, the I/O completion ports only server I/O with the named pipes, not connection administration). As the documentation to **GetQueuedCompletionStatus** mentions, an application can prevent an asynchronous I/O operation from sending completion notifications to a completion port by setting the low-order bit of the event object.

- **CServerDatabaseProtocol** (Protocol.cpp and Protocol.h in the COMMON subdirectory) is the main class that implements the server. This class is responsible for dispatching the worker threads, coordinating the I/O between the worker threads, and communicating with the clients through the **CServerNamedPipe** objects. (In case you are interested: I introduce protocol objects in the article "A Homegrown RPC Mechanism" in the MSDN Library.) This is the most interesting class for our discussion because here is where I/O completion ports are used. In the next section, we will dissect this class.
- **ServerChainedQueue** (Dbcode.cpp and Dbcode.h in the COMMON subdirectory) is a very crude and quick-and-dirty implementation of a database object. This class supports the **AddRecord**, **DeleteRecord**, and **RetrieveRecord** methods, where a record is simply a data structure consisting of two integers. There is no magic whatsoever to this class. I leave it as an exercise to the reader to replace this class with a "real" database object, for example a DAO OLE Automation server as imported with an OLE type library. The **ServerChainedQueue** class is what the server calls in to process client database commands.
- **CClientObject** (Client.cpp and Client.h in the SRV subdirectory) is a representation of a client in the server application. We will look at this class later on.
- **CSecSrvView** (Secsrvvw.cpp in the SRV subdirectory) is the MFC view class that is used to communicate with the user.

The control flow in the server application is as follows: The **CSecSrvView** object instance creates 25 instances of the **CServerNamedPipe** object (this is a purely random number that can easily be changed through the symbolic identifier MAXCLIENTCOUNT) and then dispatches a thread that continuously waits for clients to connect to a free **CServerNamedPipe** object. The view then creates one object of type **CServerDatabaseProtocol**, which we will discuss soon.

As soon as a client has connected, the view creates an instance of the **CClientObject** class that represents the client and then calls into the **CServerDatabaseProtocol::Associate** member function to tell the protocol that a new client is waiting to be serviced. **CServerDatabaseProtocol** maintains a fixed number of worker threads that use IOCPs to service client requests.

Let us look at the constructor for **CServerDatabaseProtocol** really quickly:

```
CServerDatabaseProtocol::CServerDatabaseProtocol(int iThreadCount)
{
  bActive=TRUE;
  m_iThreadCount=iThreadCount;
```

```
    if (iThreadCount>MAXTHREADCOUNT)
    m_iThreadCount=MAXTHREADCOUNT;
    DWORD id;
    m_hPort=
       ::CreateIoCompletionPort(INVALID_HANDLE_VALUE,NULL,NULL,m_iThreadCount);
    for (int iLoop=0;iLoop<m_iThreadCount;iLoop++)
     {
  // Now create the worker threads.
      m_coThreads[iLoop]=::CreateThread(NULL,0,
                                  (LPTHREAD_START_ROUTINE)WorkerThreadFunction,
                                  (void *)this,CREATE_SUSPENDED,&id);
      ::SetThreadPriority(m_coThreads[iLoop],THREAD_PRIORITY_BELOW_NORMAL);
      ::ResumeThread(m_coThreads[iLoop]);
      m_coClients[iLoop]=NULL;
     };
  };
```

To create an I/O completion port and to associate a client communication with an existing IOCP, the server application uses the system call **CreateIoCompletionPort**. The prototype for this call is as follows:

```
HANDLE CreateIoCompletionPort (HANDLE FileHandle,HANDLE ExistingCompletionPort,
   DWORD CompletionKey,DWORD NumberOfConcurrentThreads)
```

The first instance of this port is normally created with the first parameter set to INVALID_HANDLE_VALUE and the second and third parameters set to NULL. The return value from this call is a handle that a worker thread can pass to **GetQueuedCompletionStatus** to wait for an asynchronous I/O call to finish. Don't be confused—you are right when you now wonder which I/O objects could cause a working thread to return, because no I/O object has been associated with the port yet. We will discuss this in a jiffy.

The last parameter to **CreateIoCompletionPort** is important, because it tells the I/O system how many worker threads can share the I/O port. Internally, IOCPs are implemented similarly to inverse semaphores: A semaphore is an object that can be claimed by a predefined number of threads before a claiming thread blocks. An IOCP is an object on which a predefined number of blocked threads can be awakened by completed I/O calls.

In other words, if a port is created to service five threads simultaneously, and six threads are suspended on the port while seven asynchronous I/O operations complete, only five threads are awakened, and two of the completed I/O calls remain pending until one of the six threads is ready to process another call. The preceding discussion is actually a little bit simplified, and as the documentation states, the number of threads that process I/O calls on a port may at times be higher than the number you specified. Think of the *NumberOfConcurrentThreads* parameter as a kind of hint to tell the I/O system how many threads should on the average be busy processing I/O calls. Normally, your server application will dispatch a specific number (*n*) of threads, and that is also the number you pass to **CreateIOCompletionPort**.

So what should *n* be? In other words, what is a good number of threads to dispatch? Well, if your computations are CPU-bound, you should probably dispatch no more threads than there are processors in the machine you run the server application on. Passing 0 to the number of threads parameter will default to the number of processors, or you can use the **GetSystemInfo** system service to obtain the number. For I/O-bound operation, you can probably afford

a higher degree of concurrency.

In order to associate an IOCP with a communication object, you call **CreateIoCompletionPort** again, this time passing the handle of an object that must have been created to support asynchronous I/O (such as a file, a named pipe, or a socket) and the handle of an existing completion port. As soon as an IOCP is associated with at least one I/O object, a thread that called **GetQueuedCompletionStatus** with either port handle as the first parameter may be unblocked when any asynchronous I/O call on any associated object completes. We will look at code when we discuss the **CServerDatabaseProtocol::Associate** member function.

Note that as basic as IOCPs are, they are also incredibly powerful. For example, so far we have only discussed the use of IOCPs in server applications that service multiple clients. However, there is no requirement that each communication on the same port does "the same thing." For example, it is possible that a server services one "user" client that can only submit simple queries to the database and simultaneously—that is, using the same IOCP to service both clients—services one "supervisor" client who has completely different rights on the database. It is not even necessary that the objects that are associated with the same port are objects of the same type; for example, a named pipe and a file can both be associated with the same port.

But back to the sample code. As soon as the view has created an instance of a named pipe, it calls the **Associate** member function of the **CServerDatabaseProtocol** class to associate the named pipe instance with the completion port:

```
int CServerDatabaseProtocol::Associate (HANDLE hComm, int iIndex)
{
CreateIoCompletionPort (hComm,
                        m_hPort,
                        (DWORD)iIndex,
                        m_iThreadCount);

 return 1;
}
```

There are two interesting things to notice about this code, and those two things reveal a lot about the inner workings of I/O completion ports. First of all, the return value of the **CreateIoCompletionPort** call is never stored anywhere. This is, to a certain degree, sloppiness on my part—the return value of every system call should always be checked to ensure that no error has occurred—but the main reason that I don't bother to use the return value is that if the **CreateIoCompletionPort** call succeeds, the returned handle value will be the same as the handle passed into the call as the *m_hPort* parameter. In other words, there is only *one* physical I/O completion port, and there is no way to distinguish the instances of the port that correspond to the named pipe associations between one another.

This observation leads us to the second thing we need to know about I/O completion ports: Because there is no way to track what named pipe instances are associated with an IOCP at any given time, there is no way to dynamically remove objects from the port. I mentioned earlier that the key parameter that is used to determine which communication object unblocked a waiting thread can be arbitrarily chosen, but each communication object must be associated with the same key parameter as long as the I/O completion port exists. In an earlier version of my server application, I had passed pointers to **CClientObject** objects as the key parameters, which makes for a neat design (the thread function simply needs to convert the returned parameter to a **CClientObject** pointer, dereference, and use it). However, I had set up the application such that the **CClientObject** objects can be dynamically deleted and created as clients connect and disconnect. Thus, all of a sudden, "recycled" named pipe instances were associated

with different keys, which totally confused the I/O system. That's why I changed the design such that each named pipe instance is associated with a unique (and constant) identifier that can be used to look up the corresponding **CClientObject**.

The implementation of the **CServerDatabaseProtocol** class reveals what we have discussed earlier: That there is no correspondence between worker threads and client communications. Regardless of how many clients dynamically attach and detach, all the worker threads that will ever be active are created when the **CServerDatabaseProtocol** object is created. Let's look at the "magic" that a worker thread does:

```
long WINAPI WorkerThreadFunction(void *vArg)
{
 CServerDatabaseProtocol *csdp=(CServerDatabaseProtocol *)vArg;
 DWORD nBytes;
 DWORD WorkIndex;
 OVERLAPPED ovl;
 LPOVERLAPPED pOvl=&ovl;
 CClientObject *cpCurrentContext;
 BOOL b;
 while (csdp->bActive)
 {
  b = GetQueuedCompletionStatus(csdp->m_hPort,
                                &nBytes,
                                &WorkIndex,
                                &pOvl,
                                INFINITE);
  if (!b || !pOvl)
  { // Something has gone wrong here...
   GetLastError();
   continue;
  };
  cpCurrentContext=csdp->m_coClients[WorkIndex];
  if (cpCurrentContext->DispatchFn((STATE_ENUMERATOR)cpCurrentContext->m_se,&ovl)
      == CMD_CLIENT_TERMINATED)
    csdp->DeAssociate(WorkIndex);
};
 return 0;
};
```

Note that the thread function does nothing specific to a client-server interaction—all it does is dispatch to the client object! Thus, we need to look at the **CClientObject** implementation more closely to figure out what's going on—and what we will come across is something you probably heard about last in college, a so-called finite-state automaton.

Let's reiterate where the big problem is for the worker thread when dealing with I/O completion ports: Whenever an asynchronous I/O call has completed, it is easy for the thread to determine which client is responsible for the I/O—this can be derived from the key parameter as we discussed before—but it is not straightforward to determine *which* I/O the client completed. Let's look once more at the database example: A client-server interaction begins when the server waits for input from the client, asking for a command identifier. The client responds with either CMD_ADDRECORD, CMD_DELETERECORD, or CMD_RETRIEVERECORD, depending on what it wants from the server database. If the command was CMD_ADDRECORD, the server asks the client for the data to add to the database; and

if the command was CMD_DELETERECORD or CMD_RETRIEVERECORD, the server asks for the index of the record. When a request is processed, the server needs to write the result back to the client, and if the command was CMD_RETRIEVERECORD, the return data needs to be written to the client. How does the server determine what read or write request has been completed when an I/O request completes?

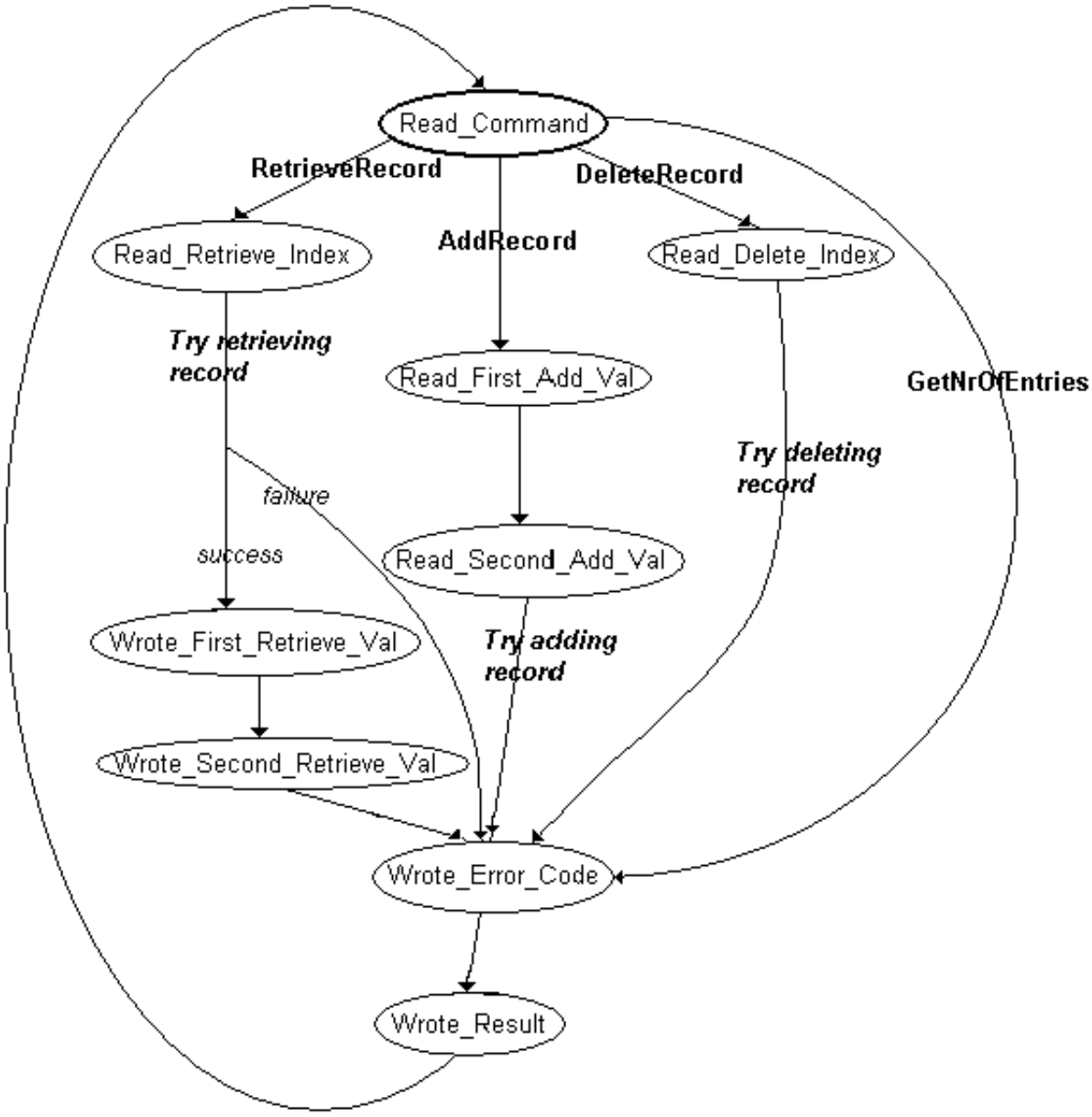

Figure 2. The database server as a finite-state automaton

We can rewrite the control flow as a diagram as depicted in Figure 2. Each circle is a state that the client-server communication can be in, and the arcs between the circles depict actions that the server performs to transform the communication from one state to the other. Note that at the end of each arc label there is an I/O call—either a read from or a write to the client, and furthermore, each I/O call is immediately followed by a state transition. Thus, with the information about the state the communication is in, the server knows exactly what to do with the I/O that just completed. Rewriting this diagram into code is easy. The **CClientObject** class has one private member variable m_se of type STATE_ENUMERATOR. That type is defined in Client.h and simply defines one symbolic constant for each state that the communication can be in:

```
typedef enum
{
 Read_Command,
 Wrote_Error_Code,
 Wrote_Result,
 Read_First_Add_Val,
 Read_Second_Add_Val,
 Read_Delete_Index,
 Read_Retrieve_Index,
 Wrote_First_Retrieve_Val,
 Wrote_Second_Retrieve_Val
} STATE_ENUMERATOR;
```

When a worker thread function is unblocked, that is, a client-server I/O call has completed, the client is first decoded from the *key* parameter. The thread function then checks to see what state the communication was in by looking at the client's m_se value. Then control is transferred to a dispatch function that resides in the client object class. This dispatch function expects as a parameter a variable of type STATE_ENUMERATOR and dispatches to an appropriate member function. Note that instead of a dispatch function, we might have implemented a function table; however, in C++, function tables of member functions are not easy to implement, so I chickened out of the gory details of C++ and wrote a dispatch function that resolves the calls at run time.

Thus, for each state that has an identifier defined in STATE_ENUMERATOR, there is a "state processing" member function in the **CClientObject** class. As a convention, I have named each of those functions just like the corresponding STATE_ENUMERATOR constant, only followed by the suffix **_Fn**. Thus, the function that processes **Read_Command** is called **CClientObject::Read_Command_Fn**. Let's look at this beast (please note, for easier understanding, that the name **Read_Command** refers to "Read" in the past tense; thus, when the communication is in this state, the command *has already been* read and stored in the m_iStatusWord member variable):

```
int WINAPI CClientObject::Read_Command_Fn(LPOVERLAPPED lpo)
{
  // Obtain the overlapped result from the
  // pipe, then branch:
  switch (m_iStatusWord)
  {
   case CMD_EXIT:
    return CMD_CLIENT_TERMINATED;
   case CMD_ADDRECORD:
    m_se=Read_First_Add_Val;
    // Now dispatch a read call to retrieve the first record.
    Read((void FAR *)&m_clElement.iSecuredElement,sizeof(int));
    return CMD_CONTINUE;
   case CMD_DELETERECORD:
    m_se=Read_Delete_Index;
    // Dispatch a read call to retrieve the delete index.
    Read((void FAR *)&m_iIndex,sizeof(int));
    return CMD_CONTINUE;
   case CMD_RETRIEVERECORD:
    m_se=Read_Retrieve_Index;
```

```
      // Dispatch a read call to retrieve the index.
      Read((void FAR *)&m_iIndex,sizeof(int));
      return CMD_CONTINUE;
     case CMD_GETENTRIES:
      m_se=Wrote_Error_Code;
      // Now call the database for the # of entries;
      // depending on the outcome, set m_iStatusWord to
      //CMD_SUCCESS or CMD_FAIL, and m_iSendValue to the
      // #of entries or the error code, respectively;
      // then dispatch a write call.
      m_iSendValue=m_cq->GetEntries();
      m_iStatusWord=CMD_SUCCESS;
      Write((void FAR *)&m_iStatusWord,sizeof(int));
      return CMD_CONTINUE;
    };
    return CMD_ERROR;
  };
```

Now the code looks at the command and takes the appropriate action. As soon as the program logic knows what the next I/O call to submit is, it sets the m_se variable to the respective value and submits the call.

It is important to mention that the client object at any point doesn't know anything about the "history" of how the communication got there. For example, as soon as the communication is in the Wrote_Error_Code state, it doesn't matter at all whether the client request that got the communication there was an **Add**, **Delete**, or **Retrieve** request; all the communication knows is that when this state was reached, somebody (that means, the function that was called for some other state) has set the m_iErrorCode variable to whatever was appropriate, and the value has been successfully written to the client. Also, at this point, the m_iStatusWord member variable has been set to the return code of whatever the operation returned, and so the state handler **Wrote_Error_Code_Fn** knows that all it needs to do is to write the m_iStatusWord value out through the communication channel and set the next state to **Wrote_Result**.

Speaking in simple terms, the m_se variable tells the code what the current I/O return from the client means and thus, how to interpret it and how to follow up on it. Speaking in terms that impress other guests on a cocktail party, the client logic implements a finite-state automaton in which each state represents one completed I/O call, and the transitions between the states represents the actions that the server takes depending on the I/O.

The big benefit of using C++ to implement the server is that we can strictly separate the I/O logic from the client interactions. In other words, as long as a client that is attached to **CServerDatabaseProtocol** with the Associate member function has an m_se variable and a **Dispatch** function, we can associate any client to the interaction without changing a thing to the **CServerDatabaseProtocol** object. We can even define **CClientObject** as a base class, derive different client types from it, and attach clients from any derived class to the **CServerDatabaseProtocol** object.

Postlude

After I had finished coding the sample application suite, I was curious to see what exactly the performance benefits were that I/O completion ports buy you. Before I coded the server using I/O completion ports, I had simply taken the server from the security sample suite, stripped the security code, and rewritten the server view to serve multiple clients using the one-thread-per-client approach. Those changes were trivial; it took me about 20 minutes to extend the single-client server to a multiple-client one. This is because, as I mentioned before, there was one worker thread

in the single-client server, and basically all I had to do to extend the server was to create one new thread for each client that connects. Thus, if the development effort is so trivial (as opposed to the hours it took me to rewrite the server to support I/O completion ports), why go through the hassle in the first place?

The client application has a number of new options (new as opposed to the "old" version of the client as published in the MSDN Library) that relate to performance; I added a mechanism to perform 1,000 database transactions and sample the results in terms of performance counter ticks. The other new option, batch processing, does 10 sets of 1,000 additions and deletions each; thus, by hitting the server hard (I opened 20 instances of the client application, each of which ran the batch processing script), I was able to obtain fairly reliable performance figures.

I ran sets of one, five, ten, and twenty clients, all running the script against both the one-thread-per-client approach and the I/O-completion-port-based server. Because the I/O-completion-port-based server is coded to create five worker threads, the greatest differences should show up where, respectively, 10 and 20 worker threads do the work in the one-thread-per-client implementation as opposed to the five worker threads. The performance—that means, the average turnaround time for a single transaction—that each client saw under a corresponding workload was pretty much identical to the one-thread-per-client and the I/O completion port based server. I attribute this finding to the fact that the database transactions in my little server are I/O bound and, therefore, do expose the problems of multiple threads very well. However, the overall turnaround time for each client to complete its test script was significantly longer when I employed the one-thread-per-client server. Also, the machine that ran the server was hardly usable at all under the load of 20 worker threads, whereas the same workload executed on the I/O-completion-port-based server made the machine that executed the server still very reasonably responsive and usable.

Summary

In a world in which more and more tasks are performed in client-server environments, the design of a good and responsive server application is absolutely crucial, and an important part of a good server application design is to work around the bottlenecks that a high client workload can impose. I/O completion ports are an essential tool for a good server application design. Whereas I/O completion ports are less straightforward to code than corresponding servers that do not use I/O completion ports, the techniques I presented in this article can be used to take the sting out of server application design.

Please note, once more, that there are other uses for I/O completion ports than the one I discussed in this article.

Bibliography

Asche, Ruediger. "Win32 Multithreading Performance." January 1996. (MSDN Library, Technical Articles)

Heller, Martin. "Tips and Tricks on Developing Killer Server Applications for Windows NT." *Microsoft Systems Journal* 10 (August 1995). (MSDN Library, Periodicals)

Vert, John. "Writing Scalable Applications for Windows NT." June 1995. (MSDN Library, Technical Articles)