**Scalable Servers with IO Completion Ports and How to Cook Them**

By **[rtybase](http://www.codeproject.com/script/Membership/View.aspx?mid=1540412)** | 1 Oct 2007

**Sponsored Links**

窗体顶端

* [Download source files - 115 KB](http://www.codeproject.com/KB/IP/IOCP_how_to_cook/echo_server.zip)

**1. Introduction**

Quite a long time has passed since I promised to write this article. Finally, it is ready. So, this article will be a continuation of the MP3 streaming server project (started with the ["Sound recording and encoding in MP3 format"](http://www.codeproject.com/KB/audio-video/RecordingAndMP3Encoding.aspx) article) and, particularly, will be dedicated to a very interesting topic: writing scalable server applications using IO Completion Ports (IOCP for further reference). I will also describe a framework which handles all the complications related to this.

Technically, the entire topic can be split into three sub-topics: thread management, memory management, and client sockets handling mechanism. Regarding the latter, I will focus on TCP sockets (something similar can be implemented for UDP sockets). Let's proceed with the details...

**2. Threading**

Typically, the usage of threading models, when writing server applications, can be split into two major categories:

* easy to implement, **one thread serves one connection/socket**
* a bit more complicated, using thread pools (which also can be: fixed size pool, timeout based pool, variable size from minimum to maximum value, etc.)

While the first model is, usually, the easiest one to implement, it isn't the best one to use when serving a huge number of connections. There are two strong arguments against this model:

* ***Context switch***, this is the process of switching between two running processes or threads of the same or different processes. It includes saving and restoring of the context of the running thread/process in such a fashion as if each thread/process would run continuously, where in fact, they may share the same CPU. Obviously, context switch takes time, and this time depends on the Operational System implementation. E.g., with Windows OS, it takes nanoseconds, while in some old versions/distributives of the Linux (e.g., Mandrake version <= 7.2), it takes ~2 milliseconds. So, 2 milliseconds multiplied by (e.g.,) 1000 threads = 2 seconds to return the context back to a thread, having just 1000 created threads, provided that all threads have the same priority. This is not acceptable and, obviously, a different model is needed.
* Typically, threads, in the *one thread serves one connection* model, do nothing most of the time. With more details, they are frozen while performing blocking I/O operations against (e.g.,) sockets. As so, having a huge number of created threads is nothing but wasting of resources.

The IOCP framework, described in this article, uses the threading framework, particularly the CSimpleThreadPool class, described in the ["Almost like Java threads"](http://www.codeproject.com/KB/IP/LikeJavaThreads.asp) article. There is nothing special with that threading framework, though I will provide a few reasons why I am using it:

* It is part of this project.
* It is very light, and still quite useful and easy to use.
* CSimpleThreadPool is a fixed size thread pool. Having a pre-created amount of threads helps in reducing the run-time time needed to create new threads as they are already created. Mind that creating a new thread takes time (e.g., the system must create a new system object (thread itself), must create a thread's stack etc.). So, this is good for performance.
* And, the main reason, CSimpleThreadPool is a thread pool with a priority queue implementation. Anticipating details described below, the IOCP queue isn't prioritizing the completion requests; all completion requests have the same priority. So, to write (and run) a very simple application that simply creates thousands of connections and close them all at once and once again and so on ... IOCP will have registered in the queue a lot of close requests and, effectively, the server will be busy with serving just close requests. This is an example of a Denial of Service attack. It's sad that IOCP isn't using a priority queue to prioritize the completion requests, but I imagine there were technical reasons not to use such a queue. However, completion requests received from the IOCP queue (all with the same priority) are passed to the CSimpleThreadPool queue, and there they are prioritized, so, close connection requests will have a lower priority than other completion requests. This is just a strategy to defend the IOCP framework, described in this article, against such an attack.

**3. Memory management**

Another problem, related to writing server applications, is correct memory management. While the quite familiar operators like new and delete (or counterparts, malloc()/free(), or any other low level memory Windows SDK API) are very trivial to use, there is a problem related to them, when talking about applications that should run 24 hours per day, 7 days per week, 365 days per year.

The problem is the so named ***memory fragmentation***. These operators (APIs) are designed to allocate continuous blocks of memory, and if the application is performing new/delete very frequently and with different block sizes, it will end up with situations when new returns NULL (maybe constantly), despite the fact that the application has enough free space in the heap. This may happen because ... right ... there is no continuous free space of the requested size, in other words: we have a *memory fragmentation* and, sooner or later, the server will fail.

There are a few workarounds to the *memory fragmentation* problem:

* Implementing garbage collection（GC）, which also performs the de-fragmentation.
* Custom memory management designed to avoid fragmentation with, potential, overriding of the new/delete, just to make the final solution elegant.
* Using the new operator (malloc()) to allocate blocks of fixed size. Well, memory still tends to fragment, but "holes" are of fixed size as well (well, or of a factor multiplied by the fixed size). This is a very easy mechanism to implement. However, if the real memory needs vary from (e.g.,) 1KB to 1MB, allocating fixed size blocks of 1MB, even if only 1KB is needed, is nothing but waste of memory.
* Pre-allocated memory area. In cases when it is possible to estimate the memory required for serving one single connection, multiply that value by the maximum number of allowed connections (which can be a configuration parameter). With the obtained value, allocate a huge block of that size and "work" within that area. Another very easy to use mechanism, which works fine when ... such estimation is possible to perform.
* Re-using of the allocated blocks. The idea is very simple; once a block is allocated, don't delete it when it is not in need, just mark it as un-used and ... re-use it next time. This may be seen as a *memory leak* where, in fact, it isn't; it is just a strategy to handle *memory fragmentation*.

Below is a template class (for the complete source code, see the *mem\_manager.h* file in the attached sources) which handles the "Pre-allocated memory area" and "Re-using of the allocated blocks" (all in one) mechanisms described above:

*// A template class allowing re-using of the memory blocks.*

template<class T>

class QueuedBlocks {

private:

QMutex m\_qMutex;

set< T\* > m\_quBlocks;

vector< T\* > m\_allBlocks;

public:

QueuedBlocks(int nInitSize = 1):

m\_qMutex(), m\_quBlocks(), m\_allBlocks() {...};

*// get a free block from the queue, if one cannot be found*

*// then NULL is returned*

T\* GetFromQueue() {...};

*// get a free block from the queue, if one cannot be found*

*// then a new one is created*

T\* Get() {...};

*// Release the used block, place it*

*// back to the queue. For performance reason,*

*// we assume that the block was previously taken*

*// from the queue.*

void Release(T\* t) {...};

*// Return all the blocks ever allocated.*

vector<T\*> \*GetBlocks() {...};

~QueuedBlocks() {...};

};

So, with the constructor, we pre-allocate a number of requested objects, and:

* If just the "Pre-allocated memory area" mechanism is needed, use just the GetFromQueue() and Release() methods, so it will work "within the allocated space".
* If "Re-using of the allocated blocks" is needed (well, together with "Pre-allocated memory area"), use the Get() and Release() methods, so if pre-allocated space is not sufficient, it will be extended.

Since this is a generic template, it allows defining queued blocks of different classes/structures. The only problem with this implementation is, each particular class must have a default (or no arguments) constructor and a Clear() method. The goal of the Clear() is to perform the internal clean-up, at the end, when blocks are about to be re-used.

Another useful template class is (also defined in the *mem\_manager.h*):

http://www.codeproject.com/images/minus.gif Collapse

*// A template class used for providing free blocks as well*

*// as releasing unnecessary ones. Uses "QueuedBlocks" which*

*// allows reusing of the blocks.*

template<class T>

class StaticBlocks {

private:

static QueuedBlocks<T> \*blocks;

public:

static void Init(int nSize = 1) {...};

static T \*Get() {...};

static void Release(T \*b) {...};

};

It represents an adapter of the QueuedBlocks, but has all methods defined as static, which is quite useful when it is necessary to share blocks across different classes.

**4. A few facts about IOCP**

If you are reading this article, you probably know what the benefit of using IOCP is. If you don't know, then ... IOCP is the best mechanism to write scalable server applications, capable of handling thousands of client connections with just a few threads. How is this accomplished? Well, by using a system object named "I/O Completion Port" which "**lets an application receive notification of the completion of asynchronous I/O operations**". So, every I/O operation is asynchronous (completes immediately), the application (or its threads) is notified about the completion of I/O operations and, so, the application (or the application's threads) has "enough time" to perform any other operations (rather than waiting for the I/O completion, as usually happens in the "one thread serves one connection/socket" model). Let's count the facts we know about IOCP.

First of all, in order to make sockets to benefit from IOCP, overlapped I/O versions of the Winsock API must be used.

WSASend(...);

WSARecv(...);

Secondly, a socket must be configured to support overlapped API.

nSock = WSASocket(..., ..., ..., ..., ..., **WSA\_FLAG\_OVERLAPPED**);

We also need to have an IOCP handle created.

hIocp = CreateIoCompletionPort( INVALID\_HANDLE\_VALUE, NULL, 0, 0 );

We need to associate the socket with IOCP.

CreateIoCompletionPort( (HANDLE) nSock, hIocp, ..., ... );

Now, when using overlapped I/O operations against the socket, they will be registered in the queue of IOCP. Again, this registration happens only when performing I/O operations (!!!); otherwise, this registration **will not** be performed, despite the fact that the socket is associated with IOCP. Upon completion of an overlapped I/O operation, it is returned (in fact, just details about the operation, but ... it depends on the few tricks, more details later) from the IOCP queue with the following call:

GetQueuedCompletionStatus( hIocp, &dwBytesTransferred, ..., lpOverlapped, dwTimeout );

So, at this point, we know that an I/O operation performed against the socket is successfully completed. Considering these facts, let's proceed with the implementation of the few classes that will support all these.

**5. Client/Server sockets**

First of all, let's analyze the ClientSocket template class (for the complete source code, see the *socket\_client.h* file in the attached sources):

*// A class wrapping basic client socket's operations*

template<class T>

class ClientSocket {

private:

SOCKET m\_ClientSock;

volatile unsigned int m\_nSession;

QMutex m\_qMutex;

struct sockaddr\_in m\_psForeignAddIn;

volatile bool m\_blnIsBusy;

T m\_objAttachment;

protected:

public:

ClientSocket(): m\_objAttachment(), m\_qMutex() {...}

~ClientSocket() {...}

*// Is the object assigned a socket.*

bool IsBusy() {...};

*// Method is used by 'QueuedBlocks' from*

*// "mem\_manager.h".*

void Clear() {...}

*// Returns the socket associated with the object.*

SOCKET GetSocket() {...}

*// Returns the session of the current client socket.*

*// Technically, this is the counter of how may times*

*// current instance was pooled, but really, this is used*

*// to check if session of operation (MYOVERLAPPED)*

*// is == with the session of the socket.*

unsigned int GetSession() {...}

*// Internally used by this implementation. Don't call it.*

void Lock() {...}

*// Internally used by this implementation. Don't call it.*

void UnLock() {...}

*// Associate the object with a socket.*

int Associate(SOCKET sSocket, struct sockaddr\_in \*psForeignAddIn) {...}

*// Returns the attachment of the object.*

*// Use the attachment to associate the socket with any other*

*// additional information required.*

T\* GetAttachment() {...}

*// Write data to the socket,*

*// uses overlapped style.*

int WriteToSocket(char \*pBuffer, DWORD buffSize) {...}

*// Read data from the socket,*

*// uses overlapped style.*

int ReadFromSocket(char \*pBuffer, DWORD buffSize) {...}

*// Closes the socket associated with this instance*

*// of the object and performs clean-up.*

void CloseSocket() {...}

};

There are a few things to mention about this class:

* It is designed just to work together with the ServerSocket template class (more details below). Using it standalone won't be very useful.
* It encapsulates the overlapped I/O version of the Winsock API.
* It is a generic template class to support "attachments" (see the GetAttachment() method). While the basic functionality of theClientSocket is pretty much standard, attachments allow extending the functionality of the ClientSocket class (e.g., keeping the current state of the socket, containing data buffers for read/write etc.). The only limitation with the class/structure used as an attachment is: it should have a default (no arguments) constructer and a Clear() method (to perform internal cleaning). There are a few other requirements, but they will be described later.
* It is designed to allow the ServerSocket re-using the already created instances; as we will see later, the ServerSocket contains aQueuedBlocks<ClientSocket<T> > (the ClientSocket must implement Clear()). That is why the physical socket handler is not passed to the constructed, rather it is associated with a free (IsBusy()) instance (via Associate()).

Secondly, let's analyze the ServerSocket template class (for the complete source code, see the *socket\_servert.h* file in the attached sources):

*// A class for handling basic server socket operations.*

template<class T>

class ServerSocket {

private:

*// Server socket.*

SOCKET m\_ServSock;

*// Pool of client sockets.*

QueuedBlocks<ClientSocket<T> > m\_SockPool;

protected:

public:

*// Set the size of the socket pool to the maximum number of accepted*

*// client connections.*

ServerSocket(unsigned int nPort, unsigned int nMaxClients,

bool blnCreateAsync = false,

bool blnBindLocal = true): m\_SockPool(nMaxClients) {...}

*// Returns a pointer to the ClientSocket instance, bind*

*// to the accepted client socket (incoming connection).*

ClientSocket<T>\* Accept() {...}

*// Release the client socket, will try closing connection and*

*// will place it back to the pool.*

void Release(ClientSocket<T> \*sock) {...}

vector<ClientSocket<T> \*> \*GetPool() {...}

~ServerSocket() {...}

};

A few details about the ServerSocket template class:

* It is a generic template class ... just because it operates with the generic ClientSocket (again, the whole point is in the "attachment").
* It works with a fixed number of ClientSocket instances. That's why Accept(), internally, calls the GetFromQueue() method of them\_SockPool (not just Get()). This simulates the fact that the Server supports a maximum number of active connections it can serve at the same time. The maximum number of active connections is dictated by the nMaxClients parameter, passed to the constructor.
* The Accept() method return a ready to use instance of the ClientSocket from the m\_SockPool (or NULL). The instance is associated with the physical socket handle of the accepted incoming connection. The Release() method is used to close the physical socket handler (if not closed yet) and push the ClientSocket instance back to the m\_SockPool, in order to re-use it later.
* The GetPool() method wraps the GetBlocks() method of the m\_SockPool. It simply returns all the instances of the ClientSockets registered in the pool (absolutely all, even those not assigned with a physical socket handler). This method is invoked by the ServerServicewhen passing the pool to the CTimeOutChecker, but ... more details later.
* The constructor of the class, internally, performs all the required operations to create the physical server socket handler, bind it to the specified port (passed to the constructor with nPort) and IP address, and set the socket in listen mode (see the details inside the code). If one of these operations fails, an exception is thrown.

**6. The *MYOVERLAPPED* structure**

OVERLAPPED (not MYOVERLAPPED), is a very important structure to the overlapped I/O version of the Winsock API and IOCP. It is used by the IOCP system object (not only that, but ... I will skip the unnecessary details) to "track" the states of the I/O operations. However, OVERLAPPED itself is quite un-useful. That is why something more advanced is required, with a little trick, of course (the complete definition is in the *socket\_client.h*).

*// Overlapped structure used with this implementation of the IOCP.*

*// Only the "Overlapped" member is used in context with the*

*// standard IOCP API, anything else is just to help this particular*

*// implementation.*

typedef struct {

*// It is very important "Overlapped" to be the very first*

*// member of the structure because pointer pointing to*

*// "this" is == to the address of the first member.*

*// This makes usage of the MYOVERLAPPED equivalent*

*// to the OVERLAPPED in context of the standard IOCP API (just a trick).*

OVERLAPPED Overlapped;

*// A structure that points to the data packet and size of the data*

*// packet associated with current overlapped operation.*

WSABUF DataBuf;

*// Operation type associated with current overlapped operation.*

IO\_SOCK\_MODES OperationType;

*// Internally used. If nSession != ClientSocket::m\_nSession*

*// that means that socket was closed (recently) and*

*// operation retrieved from IOCP queue or TaskPool queue*

*// is not relevant, to the current session of the socket.*

volatile unsigned int nSession;

*// Method is used by 'QueuedBlocks' from*

*// "mem\_manager.h".*

void Clear() {...}

} MYOVERLAPPED;

So, where is the trick? The trick is that the Overlapped member is the very first in the MYOVERLAPPED structure. As so, the memory address of an instance of the MYOVERLAPPED will always be equal to the address of the first member of the structure (in this case, Overlapped). Therefore, passing (WSASend()/WSARecv()) or receiving (GetQueuedCompletionStatus()) the pointer to a MYOVERLAPPED structure (with the correct casting, of course) is the same as if we were using the native OVERLAPPED.

This is just a trick, with a few benefits:

* With MYOVERLAPPED, we always know the type of the operation which successfully completed (e.g., READ or WRITE) via the OperationTypemember. Due to this, I take the risk to define an instance of the MYOVERLAPPED as an "operation" (for simplicity).
* With the DataBuf member, we know the details about the data buffer of the operation which successfully completed (see theWriteToSocket() and ReadFromSocket() methods of the ClientSocket).
* Well, the nSession member is another trick used by the IOCP framework described in this article. ClientSocket has a session (see section 5 above), and MYOVERLAPPED has a session. The session of a ClientSocket instance is increased every time ServerSocket::Release() is invoked (which means the instance is ready for re-use). When invoking (e.g.,) the WriteToSocket() or ReadFromSocket() methods, inside, they set the current session of the ClientSocket instance to the session of the MYOVERLAPPED instance used with the operation. When receiving the completed operation with GetQueuedCompletionStatus(), and if the session of the returned operation is different from the session of the ClientSocket instance which initiated the operation, then the operation is no more relevant, because the ClientSocketinstance succeeded to close and, maybe, re-used while the operation (e.g.,) got stuck in the IOCP queue. So, inside the whole code, you will see a few of such checks.

And, just because the MYOVERLAPPED structures are shared across different classes, the following (as described in section 3 above) is important:

*// a pool of the MYOVERLAPPED structures*

typedef StaticBlocks<MYOVERLAPPED> Overlapped;

QueuedBlocks<MYOVERLAPPED> \*Overlapped::blocks = NULL;

**7. The *IOCPSimple* class**

One of the key template classes of the framework described in this article is IOCPSimple (for the complete source code, see the *iocp.h* file in the attached sources).

*// A template class (wrapper) to handle basic operations*

*// related to IO Completion Ports (IOCP).*

template<class T>

class IOCPSimple {

private:

HANDLE m\_hIocp;

DWORD m\_dwTimeout;

protected:

*// Useful method to queue a socket (handler in fact) in the IOCP's*

*// queue, without requesting any 'physical' I/O operations.*

BOOL PostQueuedCompletionStatus( ClientSocket<T>\* sock,

MYOVERLAPPED \*pMov ) {...}

public:

~IOCPSimple() {:}

IOCPSimple( DWORD dwTimeout = 0 ) {...}

*// Method associates Socket handler with the handler*

*// of the IOCP. It is important to do this association*

*// first of all, once Socket is accepted and is valid.*

*// In fact, this method registers socket with the IOCP,*

*// so IOCP will monitor every I/O operation happening*

*// with the socket.*

void AssociateSocket( ClientSocket<T>\* sock ) {...}

*// Queue in the IOCP's queue with the status 'pending'.*

*// This is a fictive status (not real). Method is useful in*

*// cases when packets/data buffers to send via socket are yet*

*// not ready (e.g. due some internal, time consuming, calculations)*

*// and rather than keeping socket somewhere, in separate structures,*

*// place it in the queue with status pending. It will be returned*

*// shortly with the "GetQueuedCompletionStatus" call anyway.*

BOOL SetPendingMode( ClientSocket<T> \*sock ) {...}

*// Queue in the IOCP's queue with the status 'close'.*

*// This is a fictive status (not real). Useful only*

*// when close socket cases should be treated/handled*

*// within the routine, handling I/O completion statuses,*

*// and just to not spread the code across different*

*// modules/classes.*

BOOL SetCloseMode( ClientSocket<T> \*sock,

MYOVERLAPPED \*pMov = NULL ) {...}

*// Queue in the IOCP's queue with the status 'accept'.*

*// This is a fictive status (not real). When a new*

*// client connection is accepted by the server socket,*

*// method is used to acknowledge the routine, handling*

*// I/O completion statuses, of this. Same reason as per*

*// 'close' status, just to have a unique piece of code*

*// handling all the possible statuses.*

BOOL SetAcceptMode( ClientSocket<T> \*sock ) {...}

*// This method is just a wrapper. For more details,*

*// see MSDN for "GetQueuedCompletionStatus". The only*

*// difference is, method is adjusted to handle*

*// 'ClientSocket<T>' (as registered with 'AssociateSocket(...)')*

*// rather than pointer to the 'DWORD'.*

BOOL GetQueuedCompletionStatus( LPDWORD pdwBytesTransferred,

ClientSocket<T> \*\*sock, MYOVERLAPPED \*\*lpOverlapped ) {...}

};

So, the class is really just a simple wrapper of the API functions associated with IOCP and, yes, it is a generic template class, due to theClientSocket ("attachment" stuff). Most of the details related to the IOCP logic, we captured in section 4 (above); however, there is something additional to write here.

The third parameter of the following function:

CreateIoCompletionPort( ..., ..., Key, ... );

is expected to be a DWORD, in the API it is named *Key*. This *Key* is returned, whenever an I/O operation completes, to the third parameter of the following function:

GetQueuedCompletionStatus( ..., ..., &Key, ..., ... );

The *Key* can be anything that fits (using casting, of course) in a DWORD. As an example, this can be a pointer to an object. What if we use, as a *Key*, the address (pointer) of the instance of the ClientSocket against which I/O operations will be performed? Right, this will allow us to know not just the operation that completed, but also the instance of the ClientSocket against which the I/O operation was applied. Here is another trick, and that is what the AssociateSocket() and GetQueuedCompletionStatus() methods of the IOCPSimple do.

**8. The *SetPendingMode()* method of the *IOCPSimple* class**

I should write a few words about the SetPendingMode() method of the IOCPSimple template class. It is a quite useful and dangerous method at the same time. Consider a case when your application is reading data, in chunks, from an external source and sending them to an instance of theClientSocket. If the external source is not ready to return the next chunks (for any reason), then nothing will be sent to the ClientSocket, and as such, no I/O operations will be queued in the IOCP (as described in section 4). So, the state of the ClientSocket is undefined, and it is very possible to "lose" such a socket. Rather than implementing a cache storage to "host such sockets and treat them later when data will be available", use the SetPendingMode() method. This method will push the socket to the IOCP queue and, in a little while, the socket will be returned with theOnPending() event of the ISockEvent (more details in the next section). Very possible that, at that time, the next data chunks of the external source will be ready for sending (if not, use the SetPendingMode() again).

This is very useful; however, mind that SetPendingMode() just pushes the socket to the IOCP queue without actually registering any I/O operation. It is very possible that, while pushing the socket to the IOCP queue in this way, the socket may be closed by the remote end. Unfortunately, IOCP will not handle such an "unexpected" close until a physical I/O operation will be performed against the socket. Such cases are very well "filtered" by theCTimeOutChecker (more details in the next section ... so, don't underestimate it), and sooner or later (depends on the passed timeout value), such a socket will be "rejected" with the OnClose() event of the ISockEvent, provided that the timeout value is greater than zero (andCTimeOutChecker is used). Consider yourself warned.

**9. The *CTimeOutChecker* class**

This template class (template due to the "attachment" stuff), defined in the *server\_service.h* file, has as a role to check every active connection against time-out cases. A time-out case occurs when the elapsed time, since the last I/O operation, is greater than the nTimeOutValue value passed to the constructor.

*// A template class designed for verifying client sockets*

*// against time-out cases. Time-out case = if no I/O actions*

*// happen with a socket during a configured number of*

*// seconds. This implies that the attachment of the*

*// client socket must implement "GetTimeElapsed()"*

*// and "ResetTime(...)" methods.*

template<class T>

class CTimeOutChecker: public IRunnable {

private:

unsigned int m\_nTimeOutValue;

vector<ClientSocket<T> \*> \*m\_arrSocketPool;

IOCPSimple<T> \*m\_hIocp;

protected:

*// method checks sockets to detect time-out cases*

virtual void run() {...}

public:

*// Constructor of the template class. Requires:*

*// - a pointer to a collection of pointers to client sockets.*

*// This is the collection of sockets to be checked against*

*// time-out cases.*

*// - a pointer to the IO completion port object responsible*

*// for checking I/O events against passed collection of*

*// client sockets.*

*// - value of the time-out, in seconds.*

CTimeOutChecker( vector<ClientSocket<T> \*> \*arrSocketPool,

IOCPSimple<T> \*hIocp, unsigned int nTimeOutValue ) {...}

*// Nothing to destruct as inside the template class*

*// we keep/use just pointers obtained from external*

*// sources.*

~CTimeOutChecker() {};

};

The class is used by the ServerService template class (described below) and, as such, its run() method will be executed by a thread of the thread pool (CSimpleThreadPool) associated with the ServerService. The ServerService also takes care to pass the socket pool (of the associated ServerSocket, via GetPool() methods) to the (instance of) CTimeOutChecker, so CTimeOutChecker knows what sockets "to examine".

If a time-out case occurs, then the relevant (instance of) ClientSocket is passed to the IOCP queue with a status "close". IOCP, sooner or later, will return it from the queue, and will "pass" it to the ServerService. The ServerService will create an "event task", and will pass it to theCSimpleThreadPool. The "event task", which will be executed by a thread of the CSimpleThreadPool, will "fire" the OnClose() "event" of the (implementation of) ISockEvent and will close the socket. That's how it works.

Additionally, CTimeOutChecker requires the "attachment" to implement two additional methods: GetTimeElapsed() and ResetTime( bool ). So, technically, the behaviour of the CTimeOutChecker is controlled by the implementation of these two methods:

* If GetTimeElapsed() returns zero for a particular (attachment of the) ClientSocket, then the time-out case is not applicable to theClientSocket. This is a way to disable time-out check for a particular socket.
* Despite the fact that the implementation of the ResetTime( bool ) can be arbitrary (depends on the developer, but must follow the signature), the actual requirements are:
  + Invoking ResetTime( true ) (with true!) must force any subsequent call of the GetTimeElapsed() to return zero. This allows disabling time-out check, for a particular socket, at some point (when required). E.g., when writing a Web (HTTP) server, it makes sense to enable time-out check only while waiting for the complete HTTP request from the ClientSocket. Once the complete HTTP request is received, it makes sense to disable time-out check for the relevant ClientSocket.
  + Invoking ResetTime( false ) should force any subsequent call of the GetTimeElapsed() to return the real elapsed time, in seconds, since the last invocation of the ResetTime( false ). In other words, it should register the current date-time, andGetTimeElapsed() should return the difference, in seconds, of the current and registered date-times. It is also up to the developer to invoke ResetTime( false ) when ClientSocket::WriteToSocket() or ClientSocket::ReadFromSocket() successfully completes (depends on the implementation policy of the server being developed).

What is CTimeOutChecker useful for? Well, just to close the sockets which "don't respond" for a (configured) while and thus, free up some space (mind the nMaxClients parameter of the ServerSocker class!) for the other incoming connections.

However, if this logic isn't required, there is a way to disable (entirely) the CTimeOutChecker. Just set the timeout parameter of theServerService constructor to zero.

**10. *ISockEvent* and *ServerService* classes**

Finally, let's proceed with the core of the framework. It is OK if you don't remember all the details described above, because they all are encapsulated in the logic of the ServerService template class (the reason is still the same, "attachment" stuff), though it would be great to keep sections 8 and 9 in mind. Both classes are defined in the *server\_service.h* file. Let's start with the ServerService one:

*// Well, finally, here is the template class joining all*

*// the stuff together. Considering the Aspect Oriented paradigm,*

*// this template class may be seen as an Aspect. The "individualizations"*

*// of this aspect are "ISockEvent<T>" and "T" itself. "T" is nothing*

*// else but attachment of the client socket (see ClientSocket<T> template*

*// class for more details). Implementing "ISockEvent<T>" and "T" will*

*// define the individual behaviour of this aspect.*

*// It is a composition of the IOCP, server socket, time-out checker*

*// and thread pool. Class implements the business logic that makes all*

*// these entities working together.*

template<class T>

class ServerService: public IRunnable {

private:

ServerSocket<T> m\_ServerSocket;

IOCPSimple<T> m\_hIocp;

ISockEvent<T> \*m\_pSEvent;

CTimeOutChecker<T> \*m\_TChecker;

*// thread pool which will execute the tasks*

CSimpleThreadPool \*m\_ThPool;

*// a pool of the CSockEventTask<T> objects*

QueuedBlocks<CSockEventTask<T> > m\_SockEventTaskPool;

protected:

*// This method will be executed by a thread*

*// of the task pool.*

virtual void run() {...}

public:

*// Constructor or the class.*

*// pSEvent - pointer to an instance implementing*

*// ISockEvent<T>. This instance will be used*

*// as a client socket event handler.*

*// nPort - port number to bind server socket to.*

*// nMaxClients - the maximum number of accepted (concurrent)*

*// client connections. To be passed to*

*// the server socket and also will be used*

*// as the initial size for the pool of the*

*// CSockEventTask<T> objects.*

*// nNoThreads - indicative (the real one is computed,*

*// see below) number of the threads*

*// to be created by the thread pool.*

*// timeout - the value of the time-out, in seconds.*

*// Will be passed to the time-out checker.*

*// If time-out is zero, time-out checker*

*// will not be created.*

*// blnBindLocal - see ServerSocket<T> for more details.*

*// If "true" then server socket is bind*

*// to localhost ("127.0.0.1").*

*// If "false" then server socket is bind*

*// to INADDR\_ANY ("0.0.0.0").*

ServerService( ISockEvent<T> \*pSEvent, unsigned int nPort,

unsigned int nMaxClients, unsigned int nNoThreads,

unsigned int timeout, bool blnBindLocal = true ):

m\_ServerSocket( nPort, nMaxClients, true, blnBindLocal ),

m\_hIocp( 200 ), m\_SockEventTaskPool( nMaxClients )

{...}

virtual ~ServerService() {...}

*// Start the threat pool (== all the*

*// threads in the pool).*

void start() {...}

};

That's it, no more tricks. The comments associated with the code (and code itself) should be sufficient to highlight the logic. However, I will focus on providing just one little detail about the ISockEvent template class (in fact, template interface), passed as a parameter to the constructer of theServerService.

ISockEvent is a template interface with pure virtual methods:

*// A template interface showing how the client socket event*

*// handler should look like.*

template<class T>

class ISockEvent {

public:

*// Client socket ("pSocket") is about to be closed.*

virtual void OnClose( ClientSocket<T> \*pSocket,

MYOVERLAPPED \*pOverlap,

ServerSocket<T> \*pServerSocket,

IOCPSimple<T> \*pHIocp

) = 0;

*// Client socket ("pSocket") was just accepted by*

*// the server socket (new connection with a client*

*// is created).*

virtual void OnAccept( ClientSocket<T> \*pSocket,

MYOVERLAPPED \*pOverlap,

ServerSocket<T> \*pServerSocket,

IOCPSimple<T> \*pHIocp

) = 0;

*// Client socket ("pSocket") was returned from the IOCP*

*// queue with status \_PENDING. For more details see*

*// "IOCPSimple<T>::SetPendingMode(...)". This method*

*// is invoked only if there was a call to*

*// "IOCPSimple<T>::SetPendingMode(...)", performed by a*

*// user code, internally "SetPendingMode(...)"*

*// is never called.*

virtual void OnPending( ClientSocket<T> \*pSocket,

MYOVERLAPPED \*pOverlap,

ServerSocket<T> \*pServerSocket,

IOCPSimple<T> \*pHIocp

) = 0;

*// Client socket ("pSocket") was returned from IOCP*

*// queue with status \_READ. This means that overlapped*

*// reading operation, started previously with*

*// "ClientSocket<T>::ReadFromSocket(...)", was*

*// successfully finished.*

*// - "pOverlap->DataBuf" structure points to the data*

*// buffer and buffer's size that were passed to the*

*// "ClientSocket<T>::ReadFromSocket(...)".*

*// - "dwBytesTransferred" will indicate how many*

*// bytes were read.*

virtual void OnReadFinalized( ClientSocket<T> \*pSocket,

MYOVERLAPPED \*pOverlap,

DWORD dwBytesTransferred,

ServerSocket<T> \*pServerSocket,

IOCPSimple<T> \*pHIocp

) = 0;

*// Client socket ("pSocket") was returned from IOCP*

*// queue with status \_WRITE. This means that overlapped*

*// writting operation, started previously with*

*// "ClientSocket<T>::WriteToSocket(...)", was*

*// successfully finished.*

*// - "pOverlap->DataBuf" structure points to the data*

*// buffer and buffer's size that were passed to the*

*// "ClientSocket<T>::WriteToSocket(...)".*

*// - "dwBytesTransferred" will indicate how many*

*// bytes were written.*

virtual void OnWriteFinalized( ClientSocket<T> \*pSocket,

MYOVERLAPPED \*pOverlap,

DWORD dwBytesTransferred,

ServerSocket<T> \*pServerSocket,

IOCPSimple<T> \*pHIocp

) = 0;

};

Implementing this interface (and the "attachment", of course) will define the server application. Nothing more should be done ... really! ISockEventcontains all the necessary methods (let's call them events) to follow and treat all the possible states of the socket. Let's see:

* When a new connection is accepted by the ServerSocket, the ServerSevice will make sure that the OnAccept() event is invoked. So, it is up to the developer how to treat this event. Typically, with this event, you will probably initiate pSocket->ReadFromSocket() orpSocket->WriteToSocket(), depending on the policy (the server expects the client to initiate a "conversation" or vice-versa) of the server to be developed.
* If a socket is about to be closed (remote end closed connection; CTimeOutChecker initiated this operation, or you invokepHIocp->SetCloseMode(pSocket) from within the code of an event), then the OnClose() event is invoked. Use this event to perform any required cleaning. It is OK if you don't or forget to close the socket (with pServerSocket->Release( pSocket )) within the OnClose(), because, after completion of this event, the socket will be closed anyway.
* If, within the code of an event, pHIocp->SetPendingMode(pSocket) is invoked ... right, as per section 8 above, make sure you handle this with the OnPending() event.
* If, within the code of an event (apart from the OnClose()), pSocket->ReadFromSocket() is invoked (and the returned value isn't indicating an error), the OnReadFinalized() event will be invoked once the "read" operation completes, but don't expect that the passed buffer will be completely filled to the size of the buffer. Avoid using the passed buffer during this period (from ReadFromSocket() to OnReadFinalized()). This is the requirement inherited from the IOCP API, so, we can't really do anything here.
* If, within the code of an event (apart from the OnClose()), pSocket->WriteToSocket() is invoked (and the returned value isn't indicating an error), the OnWriteFinalized() event will be invoked once the "write" operation completes. This will mean that the entire buffer was sent. Avoid using the passed buffer during this period (from WriteToSocket() to OnWriteFinalized()), for the same reason as above.
* And finally, you don't need to create additional threads, because each event is invoked within a thread of the CSimpleThreadPool associated with the ServerService. If you need more threads, pass the appropriate value with the nNoThreads parameter of the ServerServiceconstructor. And yes, make sure that each event implementation terminates sooner or later. I mean, try to avoid infinite loops within the events; otherwise, it will be problematic to stop the ServerService from running (though, still possible if checkingCThread::currentThread().isInterrupted() within the loop).

**Final note (mostly) for those planning to write server applications for multi-CPU platforms.** If the environment has *N* CPUs, then exactly *N*threads of the CSimpleThreadPool will execute the ServerService::run() method, for a better performance (as MSDN recommends). Make sure that you don't invoke pSocket->ReadFromSocket() or pSocket->WriteToSocket() multiple times from within the body of an event (try to design the application for just one successful -with no errors- invocation). This doesn't say that this framework will not be able to handle a multiple invocations scenario. The problem is, while the order of the send/receive operations is guaranteed to follow the invocation order (in the IOCP queue), the order of reporting operation completions is not guaranteed to follow the invocation order, due to the fact that GetQueuedCompletionStatus()will be invoked by multiple threads. However, if you don't care about the correct ordering, then ignore this note.

**11. A simple echo server implementation**

And, as a "proof of concept", let's implement a simple echo server which reads 7 characters from the socket and, once the buffer is filled, sends them back. The complete code is available with the sources.

**A.** First, let's define the parameters of the server and the size of the buffer:

#define BUFF\_SIZE 8

#define MAX\_CONNECTIONS 10

#define NO\_THREADS 4

#define TIME\_OUT 10

#define PORT 8080

**B.** Now, let's define the "attachment":

struct Attachment {

volatile time\_t tmLastActionTime;

char sString<BUFF\_SIZE>;

DWORD dwStringSize; *// current string's size*

Attachment() { Clear(); };

bool Commit( DWORD dwBytesTransferred ) {

DWORD dwSize = dwStringSize + dwBytesTransferred;

if ( dwBytesTransferred <= 0 ) return false;

if ( dwSize >= BUFF\_SIZE ) return false;

dwStringSize = dwSize;

sString[dwStringSize] = 0;

return true;

};

*// as requested by the API of the framework*

void Clear() { memset(this, 0, sizeof(Attachment) ); };

*// as requested by the API of the framework*

void ResetTime( bool toZero ) {

if (toZero) tmLastActionTime = 0;

else {

time\_t lLastActionTime;

time(&lLastActionTime);

tmLastActionTime = lLastActionTime;

}

};

*// as requested by the API of the framework*

long GetTimeElapsed() {

time\_t tmCurrentTime;

if (0 == tmLastActionTime) return 0;

time(&tmCurrentTime);

return (tmCurrentTime - tmLastActionTime);

};

};

**C.** Now, we need to make the template classes real classes:

typedef ClientSocket<Attachment> MyCSocket;

typedef ServerSocket<Attachment> MySSocket;

typedef IOCPSimple<Attachment> MyIOCPSimple;

typedef ISockEvent<Attachment> MyISockEvent;

typedef ServerService<Attachment> MyServerService;

**D.** Now, we implement the ISockEvent<Attachment>:

class MyISockEventHandler: public MyISockEvent {

public:

MyISockEventHandler() {};

~MyISockEventHandler() {};

*// empty method, not used*

virtual void OnClose( MyCSocket \*pSocket, MYOVERLAPPED \*pOverlap,

MySSocket \*pServerSocket, MyIOCPSimple \*pHIocp ) {};

*// empty method, not used*

virtual void OnPending( MyCSocket \*pSocket, MYOVERLAPPED \*pOverlap,

MySSocket \*pServerSocket, MyIOCPSimple \*pHIocp ) {};

virtual void OnAccept( MyCSocket \*pSocket, MYOVERLAPPED \*pOverlap,

MySSocket \*pServerSocket, MyIOCPSimple \*pHIocp ) {

int nRet;

DWORD dwSize;

char \*temp;

dwSize = BUFF\_SIZE - 1;

temp = pSocket->GetAttachment()->sString;

*// initiate the reading with OnAccept*

nRet = pSocket->ReadFromSocket( temp, dwSize );

pSocket->GetAttachment()->ResetTime( false );

if ( nRet == SOCKET\_ERROR ) {

pServerSocket->Release( pSocket );

}

};

virtual void OnReadFinalized( MyCSocket \*pSocket,

MYOVERLAPPED \*pOverlap, DWORD dwBytesTransferred,

MySSocket \*pServerSocket, MyIOCPSimple \*pHIocp ) {

int nRet;

DWORD dwSize, dwPos;

char \*temp;

*// finalize the filling of the buffer*

pSocket->GetAttachment()->Commit( dwBytesTransferred );

dwSize = BUFF\_SIZE - 1;

dwPos = pSocket->GetAttachment()->dwStringSize;

temp = pSocket->GetAttachment()->sString;

nRet = pSocket->ReadFromSocket( temp + dwPos, dwSize - dwPos );

pSocket->GetAttachment()->ResetTime( false );

if ( nRet == SOCKET\_ERROR ) {

pServerSocket->Release( pSocket );

}

else if ( nRet == RECV\_BUFFER\_EMPTY ) {

*// means that dwSize - dwPos == 0, so send the data*

*// back to the socket*

nRet = pSocket->WriteToSocket( temp, dwSize );

if ( nRet == SOCKET\_ERROR ) {

pServerSocket->Release( pSocket );

}

}

};

virtual void OnWriteFinalized( MyCSocket \*pSocket,

MYOVERLAPPED \*pOverlap, DWORD dwBytesTransferred,

MySSocket \*pServerSocket, MyIOCPSimple \*pHIocp ) {

*// clean the attachment*

pSocket->GetAttachment()->Clear();

*// and once again*

OnAccept(pSocket, NULL,pServerSocket, NULL);

};

};

**E.** And, finally:

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

{

int nRet;

MyServerService \*sService;

MyISockEventHandler \*mSockHndl;

WSAData wsData;

nRet = WSAStartup(MAKEWORD(2,2),&wsData);

if ( nRet < 0 ) {

Log::LogMessage(L"Can't load winsock.dll.\n");

return -1;

}

try {

Overlapped::Init( MAX\_CONNECTIONS );

mSockHndl = new MyISockEventHandler();

sService = new MyServerService((MyISockEvent \*) mSockHndl,

PORT, MAX\_CONNECTIONS, NO\_THREADS, TIME\_OUT, false);

sService->start();

printf("hit <enter> to stop ...\n");

while( !\_kbhit() )

::Sleep(100);

delete sService;

delete mSockHndl;

}

catch (const char \*err) {

printf("%s\n", err);

}

catch (const wchar\_t \*err) {

wprintf(L"%ls\n", err);

}

WSACleanup();

return 0;

}

**12. Microsoft VC++ 6.0 and using STL on a multi-processor environment**

As you probably noticed, in the code, I abuse using the different STL template classes like vector, set, queue etc. They work fine when compiling the application with the "default" compilation options and the application runs on a single CPU environment. However, the code will fail to work properly on a multi-CPU environment. To resolve this problem (as Microsoft suggests at [this link](http://support.microsoft.com/kb/813810)), the following must be performed:

* Open the project.
* On the ***Project*** menu, click ***Settings***.
* In the ***Configurations*** list, click ***Release***.
* Click the ***C/C++*** tab, and then click ***Code Generation*** in the ***Category*** list.
* In the ***Runtime library*** list, click ***Multi-thread (/MT)***.
* In the ***Configurations*** list, click ***Debug***.
* In the ***Runtime library*** list, click ***Multi-thread debug (/MTd)***.
* If there are other configurations in the ***Configurations*** list, set the appropriate ***Runtime library*** option for them also.
* Click ***OK***, and then rebuild the project.

**13. Conclusion**

Well, it looks easy now, isn't it? In the next article, I will describe the implementation of a few more classes, including specially customized versions of the Attachment and ISockEvent<Attachment> classes, to handle MP3 streaming. That will be the final article from this series, see you soon...窗体底端