<https://www.codeproject.com/Articles/10330/A-simple-IOCP-Server-Client-Class>

# A simple IOCP Server/Client Class

This source code uses the advanced IOCP technology which can efficiently serve multiple clients. It also presents some solutions to practical problems that arise with the IOCP programming API, and provides a simple echo client/server with file transfer.

* [**Download source v1.16 - 154.59 Kb**](https://www.codeproject.com/KB/IP/iocp_server_client/IOCP-SRC.zip)

## 1.1 Requirements

* The article expects the reader to be familiar with C++, TCP/IP, socket programming, MFC, and multithreading.
* The source code uses Winsock 2.0 and the IOCP technology, and requires:
  + Windows NT/2000 or later: Requires Windows NT 3.5 or later.
  + Windows 95/98/ME: Not supported.
  + Visual C++ .NET, or a fully updated Visual C++ 6.0.

## 1.2 Abstract

When you develop different types of software, sooner or later, you will have to deal with client/server development. To write a comprehensive client/server code is a difficult task for a programmer. This documentation presents a simple but powerful client/server source code that can be extended to any type of client/server application. This source code uses the advanced IOCP technology which can efficiently serve multiple clients. IOCP presents an efficient solution to the "one-thread-per-client" bottleneck problem (among others), using only a few processing threads and asynchronous input/output send/receive. The IOCP technology is widely used for different types of high performance servers as Apache etc. The source code also provides a set of functions that are frequently used while dealing with communication and client/server software as file receiving/transferring function and logical thread pool handling. This article focuses on the practical solutions that arise with the IOCP programming API, and also presents an overall documentation of the source code. Furthermore, a simple echo client/server which can handle multiple connections and file transfer is also presented here.

## 2.1 Introduction

This article presents a class which can be used in both the client and server code. The class uses IOCP (Input Output Completion Ports) and asynchronous (non-blocking) function calls which are explained later. The source code is based on many other source codes and articles: [1, 2, and 3].

With this simple source code, you can:

* Service or connect to multiple clients and servers.
* Send or receive files asynchronously.
* Create and manage a logical worker thread pool to process heavier client/server requests or computations.

It is difficult to find a comprehensive but simple source code to handle client/server communications. The source codes that are found on the net are either too complex (20+ classes), or don’t provide sufficient efficiency. This source code is designed to be as simple and well documented as possible. In this article, we will briefly present the IOCP technology provided by Winsock API 2.0, and also explain the thorny problems that arise while coding and the solution to each one of them.

## 2.2 Introduction to asynchronous Input/Output Completion Ports (IOCP)

A server application is fairly meaningless if it cannot service multiple clients at the same time, usually asynchronous I/O calls and multithreading is used for this purpose. By definition, an asynchronous I/O call returns immediately, leaving the I/O call pending. At some point of time, the result of the I/O asynchronous call must be synchronized with the main thread. This can be done in different ways. The synchronization can be performed by:

* Using events - A signal is set as soon as the asynchronous call is finished. The disadvantage of this approach is that the thread has to check or wait for the event to be set.
* Using the GetOverlappedResult function - This approach has the same disadvantage as the approach above.
* Using Asynchronous Procedure Calls (or APC) - There are several disadvantages associated with this approach. First, the APC is always called in the context of the calling thread, and second, in order to be able to execute the APCs, the calling thread has to be suspended in the so called alterable wait state.
* Using IOCP - The disadvantage of this approach is that there are many practical thorny programming problems that must be solved. Coding IOCP can be a bit of a hassle.

### 2.2.1 Why using IOCP?

By using IOCP, we can overcome the "one-thread-per-client" problem. It is commonly known that the performance decreases heavily if the software does not run on a true multiprocessor machine. Threads are system resources that are neither unlimited nor cheap.

IOCP provides a way to have a few (I/O worker) threads handle multiple clients' input/output "fairly". The threads are suspended, and don't use the CPU cycles until there is something to do.

## 2.3 What is IOCP?

We have already stated that IOCP is nothing but a thread synchronization object, similar to a semaphore, therefore IOCP is not a sophisticated concept. An IOCP object is associated with several I/O objects that support pending asynchronous I/O calls. A thread that has access to an IOCP can be suspended until a pending asynchronous I/O call is finished.

## 3 How does IOCP work?

To get more information on this part, I referred to other articles. [1, 2, 3, see References.]

While working with IOCP, you have to deal with three things, associating a socket to the completion port, making the asynchronous I/O call, and synchronization with the thread. To get the result from the asynchronous I/O call and to know, for example, which client has made the call, you have to pass two parameters: the CompletionKeyparameter, and the OVERLAPPED structure.

## 3.1 The completion key parameter

The first parameter, the CompletionKey, is just a variable of type DWORD. You can pass whatever unique value you want to, that will always be associated with the object. Normally, a pointer to a structure or a class that can contain some client specific objects is passed with this parameter. In the source code, a pointer to a structure ClientContext is passed as the CompletionKey parameter.

## 3.2 The OVERLAPPED parameter

This parameter is commonly used to pass the memory buffer that is used by the asynchronous I/O call. It is important to note that this data will be locked and is not paged out of the physical memory. We will discuss this later.

## 3.3 Associating a socket with the completion port

Once a completion port is created, the association of a socket with the completion port can be done by calling the function CreateIoCompletionPort in the following way:

Hide   Copy Code

BOOL IOCPS::AssociateSocketWithCompletionPort(SOCKET socket,

HANDLE hCompletionPort, DWORD dwCompletionKey)

{

HANDLE h = CreateIoCompletionPort((HANDLE) socket,

hCompletionPort, dwCompletionKey, m\_nIOWorkers);

return h == hCompletionPort;

}

## 3.4 Making the asynchronous I/O call

To make the actual asynchronous call, the functions WSASend, WSARecv are called. They also need to have a parameter WSABUF, that contains a pointer to a buffer that is going to be used. A rule of thumb is that normally when the server/client wants to call an I/O operation, they are not made directly, but is posted into the completion port, and is performed by the I/O worker threads. The reason for this is, we want the CPU cycles to be partitioned fairly. The I/O calls are made by posting a status to the completion port, see below:

Hide   Copy Code

BOOL bSuccess = PostQueuedCompletionStatus(m\_hCompletionPort,

pOverlapBuff->GetUsed(),

(DWORD) pContext, &pOverlapBuff->m\_ol);

## 3.5 Synchronization with the thread

Synchronization with the I/O worker threads is done by calling the GetQueuedCompletionStatus function (see below). The function also provides the CompletionKey parameter and the OVERLAPPED parameter (see below):

Hide   Copy Code

BOOL GetQueuedCompletionStatus(

HANDLE CompletionPort, *// handle to completion port*

LPDWORD lpNumberOfBytes, *// bytes transferred*

PULONG\_PTR lpCompletionKey, *// file completion key*

LPOVERLAPPED \*lpOverlapped, *// buffer*

DWORD dwMilliseconds *// optional timeout value*

);

## 3.6 Four thorny IOCP coding hassles and their solutions

There are some problems that arise while using IOCP, some of them are not intuitive. In a multithreaded scenario using IOCPs, the control flow of a thread function is not straightforward, because there is no relationship between threads and communications. In this section, we will represent four different problems that can occur while developing client/server applications using IOCPs. They are:

* The WSAENOBUFS error problem.
* The package reordering problem.
* The access violation problem.

### 3.6.1 The WSAENOBUFS error problem

This problem is non intuitive and difficult to detect, because at first sight, it seems to be a normal deadlock or a memory leakage "bug". Assume that you have developed your server and everything runs fine. When you stress test the server, it suddenly hangs. If you are lucky, you can find out that it has something to do with the WSAENOBUFS error.

With every overlapped send or receive operation, it is possible that the data buffer submitted will be locked. When memory is locked, it cannot be paged out of physical memory. The operating system imposes a limit on the amount of memory that can be locked. When this limit is reached, the overlapped operations will fail with the WSAENOBUFS error.

If a server posts many overlapped receives on each connection, this limit will be reached when the number of connections grow. If a server anticipates handling a very high number of concurrent clients, the server can post a single zero byte receive on each connection. Because there is no buffer associated with the receive operation, no memory needs to be locked. With this approach, the per-socket receive buffer should be left intact because once the zero-byte receive operation is completed, the server can simply perform a non-blocking receive to retrieve all the data buffered in the socket's receive buffer. There is no more data pending when the non-blocking receive fails with WSAEWOULDBLOCK. This design would be for the one that requires the maximum possible concurrent connections while sacrificing the data throughput on each connection. Of course, the more you know about how the clients interact with the server, the better. In the previous example, a non-blocking receive was performed once the zero-byte receive completes retrieving the buffered data. If the server knows that clients send data in bursts, then once the zero-byte receive is completed, it may post one or more overlapped receives in case the client sends a substantial amount of data (greater than the per-socket receive buffer that is 8 KB by default).

A simple practical solution to the WSAENOBUFS error problem is in the source code provided. We perform an asynchronous WSARead(..) (see OnZeroByteRead(..)) with a zero byte buffer. When this call completes, we know that there is data in the TCP/IP stack, and we read it by performing several asynchronous WSARead(..)with a buffer of MAXIMUMPACKAGESIZE. This solution locks physical memory only when data arrives, and solves the WSAENOBUFS problem. But this solution decreases the throughput of the server (see Q6 and A6 in section 9 F.A.Q).

### 3.6.2 The package reordering problem

This problem is also being discussed by [3]. Although committed operations using the IO completion port will always be completed in the order they were submitted, thread scheduling issues may mean that the actual work associated with the completion is processed in an undefined order. For example, if you have two I/O worker threads and you should receive "byte chunk 1, byte chunk 2, byte chunk 3", you may process the byte chunks in the wrong order, namely, "byte chunk 2, byte chunk 1, byte chunk 3". This also means that when you are sending the data by posting a send request on the I/O completion port, the data can actually be sent in a reordered way.

This can be solved by only using one worker thread, and committing only one I/O call and waiting for it to finish, but if we do this, we lose all the benefits of IOCP.

A simple practical solution to this problem is to add a sequence number to our buffer class, and process the data in the buffer if the buffer sequence number is in order. This means that the buffers that have incorrect numbers have to be saved for later use, and because of performance reasons, we will save the buffers in a hash map object (e.g., m\_SendBufferMap and m\_ReadBufferMap).

To get more information about this solution, please go through the source code, and take a look at the following functions in the IOCPS class:

* GetNextSendBuffer (..) and GetNextReadBuffer(..), to get the ordered send or receive buffer.
* IncreaseReadSequenceNumber(..) and IncreaseSendSequenceNumber(..), to increase the sequence numbers.

### 3.6.3 Asynchronous pending reads and byte chunk package processing problem

The most common server protocol is a packet based protocol where the first X bytes represent a header and the header contains details of the length of the complete packet. The server can read the header, work out how much more data is required, and keep reading until it has a complete packet. This works fine when the server is making one asynchronous read call at a time. But if we want to use the IOCP server's full potential, we should have several pending asynchronous reads waiting for data to arrive. This means that several asynchronous reads complete out of order (as discussed before in section 3.6.2), and byte chunk streams returned by the pending reads will not be processed in order. Furthermore, a byte chunk stream can contain one or several packages and also partial packages, as shown in figure 1.

***Figure 1. The figure shows how partial packages (green) and complete packages (yellow) can arrive asynchronously in different byte chunk streams (marked 1, 2, 3).***

This means that we have to process the byte stream chunks in order to successfully read a complete package. Furthermore, we have to handle partial packages (marked with green in figure 1). This makes the byte chunk package processing more difficult. The full solution to this problem can be found in the ProcessPackage(..)function in the IOCPS class.

### 3.6.4 The access violation problem

This is a minor problem, and is a result of the design of the code, rather than an IOCP specific problem. Suppose that a client connection is lost and an I/O call returns with an error flag, then we know that the client is gone. In the parameter CompletionKey, we pass a pointer to a structure ClientContext that contains client specific data. What happens if we free the memory occupied by this ClientContext structure, and some other I/O call performed by the same client returns with an error code, and we transform the parameter CompletionKeyvariable of DWORD to a pointer to ClientContext, and try to access or delete it? An access violation occurs!

The solution to this problem is to add a number to the structures that contain the number of pending I/O calls (m\_nNumberOfPendlingIO), and we delete the structure when we know that there are no more pending I/O calls. This is done by the EnterIoLoop(..) function and ReleaseClientContext(..).

## 3.7 The overview of the source code

The goal of the source code is to provide a set of simple classes that handle all the hassled code that has to do with IOCP. The source code also provides a set of functions which are frequently used while dealing with communication and client/server software as file receiving/transferring functions, logical thread pool handling, etc..

***Figure 2. The figure above illustrates an overview of the***IOCP***class source code functionality.***

We have several IO worker threads that handle asynchronous I/O calls through the completion port (IOCP), and these workers call some virtual functions which can put requests that need a large amount of computation in a work queue. The logical workers take the job from the queue, and process it and send back the result by using some of the functions provided by the class. The Graphical User Interface (GUI) usually communicates with the main class using Windows messages (because MFC is not thread safe) and by calling functions or by using the shared variables.

***Figure 3. The figure above shows the class overview.***

The classes that can be observed in figure 3 are:

* CIOCPBuffer: A class used to manage the buffers used by the asynchronous I/O calls.
* IOCPS: The main class that handles all the communication.
* JobItem: A structure which contains the job to be performed by the logical worker threads.
* ClientContext: A structure that holds client specific information (status, data, etc.).

### 3.7.1 The buffer design – The CIOCPBuffer class

When using asynchronous I/O calls, we have to provide a private buffer to be used with the I/O operation. There are some considerations that are to be taken into account when we allocate buffers to use:

* To allocate and free memory is expensive, therefore we should reuse buffers (memory) which have been allocated. Therefore, we save buffers in the linked list structures given below:

Hide   Copy Code

*// Free Buffer List..*

CCriticalSection m\_FreeBufferListLock;

CPtrList m\_FreeBufferList;

*// OccupiedBuffer List.. (Buffers that is currently used)*

CCriticalSection m\_BufferListLock;

CPtrList m\_BufferList;

*// Now we use the function AllocateBuffer(..)*

*// to allocate memory or reuse a buffer.*

* Sometimes, when an asynchronous I/O call is completed, we may have partial packages in the buffer, therefore the need to split the buffer to get a complete message. This is done by the SplitBufferfunction in the CIOCPS class. Also, sometimes we need to copy information between the buffer, and this is done by the AddAndFlush(..) function in the IOCPS class.
* As we know, we also need to add a sequence number and a state (IOType variable, IOZeroReadCompleted, etc.) to our buffer.
* We also need methods to convert data to byte stream and byte stream to data, some of these functions are also provided in the CIOCPBuffer class.

All the solutions to the problems we have discussed above exist in the CIOCPBuffer class.

## 3.8 How to use the source code?

By inheriting your own class from IOCP (shown in figure 3) and using the virtual functions and the functionality provided by the IOCPS class (e.g., threadpool), it is possible to implement any type of server or client that can efficiently manage a huge number of connections by using only a few number of threads.

### 3.8.1 Starting and closing the server/client

To start the server, call the function:

Hide   Copy Code

BOOL Start(int nPort=999,int iMaxNumConnections=1201,

int iMaxIOWorkers=1,int nOfWorkers=1,

int iMaxNumberOfFreeBuffer=0,

int iMaxNumberOfFreeContext=0,

BOOL bOrderedSend=TRUE,

BOOL bOrderedRead=TRUE,

int iNumberOfPendlingReads=4);

* nPortt

Is the port number that the server will listen on. (Let it be -1 for client mode.)

* iMaxNumConnections

Maximum number of connections allowed. (Use a big prime number.)

* iMaxIOWorkers

Number of Input/Output worker threads.

* nOfWorkers

Number of logical workers. (Can be changed at runtime.)

* iMaxNumberOfFreeBuffer

Maximum number of buffers that we save for reuse. (-1 for none, 0= Infinite number)

* iMaxNumberOfFreeContext

Maximum number of client information objects that are saved for reuse. (-1 for none, 0= Infinite number)

* bOrderedRead

Make sequential reads. (We have discussed this before in section 3.6.2.)

* bOrderedSend

Make sequential writes. (We have discussed this before in section 3.6.2.)

* iNumberOfPendlingReads

Number of pending asynchronous read loops that are waiting for data.

To connect to a remote connection (Client mode nPort= -1), call the function:

Hide   Copy Code

Connect(const CString &strIPAddr, int nPort)

* strIPAddr

The IP address of the remote server.

* nPort

The port.

To close, make the server call the function: ShutDown().

For example:

Hide   Copy Code

MyIOCP m\_iocp;

if(!m\_iocp.Start(-1,1210,2,1,0,0))

AfxMessageBox("Error could not start the Client");

….

m\_iocp.ShutDown();

## 4.1 Source code description

For more details about the source code, please check the comments in the source code.

### 4.1.1 Virtual functions

* NotifyNewConnection

Called when a new connection has been established.

* NotifyNewClientContext

Called when an empty ClientContext structure is allocated.

* NotifyDisconnectedClient

Called when a client disconnects.

* ProcessJob

Called when logical workers want to process a job.

* NotifyReceivedPackage

Notifies that a new package has arrived.

* NotifyFileCompleted

Notifies that a file transfer has finished.

### 4.1.2 Important variables

Notice that all the variables have to be exclusively locked by the function that uses the shared variables, this is important to avoid access violations and overlapping writes. All the variables with name XXX, that are needed to be locked, have a XXXLock variable.

* m\_ContextMapLock;

Holds all the client data (socket, client data, etc.).

* ContextMap m\_ContextMap;
* m\_NumberOfActiveConnections

Holds the number of connected connections.

### 4.1.3 Important functions

* GetNumberOfConnections()

Returns the number of connections.

* CString GetHostAdress(ClientContext\* p)

Returns the host address, given a client context.

* BOOL ASendToAll(CIOCPBuffer \*pBuff);

Sends the content of the buffer to all the connected clients.

* DisconnectClient(CString sID)

Disconnects a client, given the unique identification number.

* CString GetHostIP()

Returns the local IP number.

* JobItem\* GetJob()

Removes a JobItem from the queue, returns NULL if there are no Jobs.

* BOOL AddJob(JobItem \*pJob)

Adds a Job to the queue.

* BOOL SetWorkers(int nThreads)

Sets the number of logical workers that can be called anytime.

* DisconnectAll();

Disconnect all the clients.

* ARead(…)

Makes an asynchronous read.

* ASend(…)

Makes an asynchronous send. Sends data to a client.

* ClientContext\* FindClient(CString strClient)

Finds a client given a string ID. OBS! Not thread safe!

* DisconnectClient(ClientContext\* pContext, BOOL bGraceful=FALSE);

Disconnects a client.

* DisconnectAll()

Disconnects all the connected clients.

* StartSendFile(ClientContext \*pContext)

Sends a file specified in the ClientContext structure, using the optimized transmitfile(..) function.

* PrepareReceiveFile(..)

Prepares the connection for receiving a file. When you call this function, all incoming byte streams are written to a file.

* PrepareSendFile(..)

Opens a file and sends a package containing information about the file to the remote connection. The function also disables the ASend(..) function until the file is transmitted or aborted.

* DisableSendFile(..)

Disables send file mode.

* DisableRecevideFile(..)

Disables receive file mode.

## 5.1 File transfer

File transfer is done by using the Winsock 2.0 TransmitFile function. The TransmitFile function transmits file data over a connected socket handle. This function uses the operating system's cache manager to retrieve file data, and provides high-performance file data transfer over sockets. These are some important aspects of asynchronous file transferring:

* Unless the TransmitFile function is returned, no other sends or writes to the socket should be performed because this will corrupt the file. Therefore, all the calls to ASend will be disabled after the PrepareSendFile(..) function.
* Since the operating system reads the file data sequentially, you can improve caching performance by opening the file handle with FILE\_FLAG\_SEQUENTIAL\_SCAN.
* We are using the kernel asynchronous procedure calls while sending the file (TF\_USE\_KERNEL\_APC). Use of TF\_USE\_KERNEL\_APC can deliver significant performance benefits. It is possible (though unlikely), however, that the thread in which the context TransmitFile is initiated is being used for heavy computations; this situation may prevent APCs from launching.

The file transfer is made in this order: the sever initializes the file transfer by calling the PrepareSendFile(..)function. When the client receives the information about the file, it prepares for it by calling the PrepareReceiveFile(..), and sends a package to the sever to start the file transfer. When the package arrives at the server side, the server calls the StartSendFile(..) function that uses the high performance TransmitFile function to transmit the specified file.

## 6 The source code example

The provided source code example is an echo client/server that also supports file transmission (figure 4). In the source code, a class MyIOCP inherited from IOCP handles the communication between the client and the server, by using the virtual functions mentioned in section 4.1.1.

The most important part of the client or server code is the virtual function NotifyReceivedPackage, as described below:

Hide   Copy Code

void MyIOCP::NotifyReceivedPackage(CIOCPBuffer \*pOverlapBuff,

int nSize,ClientContext \*pContext)

{

BYTE PackageType=pOverlapBuff->GetPackageType();

switch (PackageType)

{

case Job\_SendText2Client :

Packagetext(pOverlapBuff,nSize,pContext);

break;

case Job\_SendFileInfo :

PackageFileTransfer(pOverlapBuff,nSize,pContext);

break;

case Job\_StartFileTransfer:

PackageStartFileTransfer(pOverlapBuff,nSize,pContext);

break;

case Job\_AbortFileTransfer:

DisableSendFile(pContext);

break;};

}

The function handles an incoming message and performs the request sent by the remote connection. In this case, it is only a matter of a simple echo or file transfer. The source code is divided into two projects, IOCP and IOCPClient, which are the server and the client side of the connection.

## 6.1 Compiler issues

When compiling with VC++ 6.0 or .NET, you may get some strange errors dealing with the CFile class, as:

Hide   Copy Code

"if (pContext->m\_File.m\_hFile !=

INVALID\_HANDLE\_VALUE) <-error C2446: '!=' : no conversion "

"from 'void \*' to 'unsigned int'"

This problems can be solved if you update the header files (\*.h) or your VC++ 6.0 version, or just change the type conversion error. After some modifications, the server/client source code can be used without MFC.

## 7 Special considerations & rule of thumbs

When you are using this code in other types of applications, there are some programming traps related to this source code and "multithreaded programming" that can be avoided. Nondeterministic errors are errors that occur stochastically “Randomly”, and it is hard to reproduce these nondeterministic errors by performing the same sequence of tasks that created the error. These types of errors are the worst types of errors that exist, and usually, they occur because of errors in the core design implementation of the source code. When the server is running multiple IO working threads, serving clients that are connected, nondeterministic errors as access violations can occur if the programmer has not thought about the source code multithread environment.

#### Rule of thumb #1:

Never read/write to the client context (e.g., ClientContext) with out locking it using the context lock as in the example below. The notification function (e.g., Notify\*(ClientContext \*pContext)) is already “thread safe”, and you can access the members of ClientContext without locking and unlocking the context.

|  |  |
| --- | --- |
| Hide   Copy Code  *//Do not do it in this way*  *// …*  If(pContext->m\_bSomeData)  pContext->m\_iSomeData=0;  *// …* | Hide   Copy Code  *// Do it in this way.*  *//….*  pContext->m\_ContextLock.Lock();  If(pContext->m\_bSomeData)  pContext->m\_iSomeData=0;  pContext->m\_ContextLock.Unlock();  *//…* |

Also, be aware that when you are locking a Context, other threads or GUI would be waiting for it.

#### Rule of thumb #2:

Avoid or "use with special care" code that has complicated "context locks" or other types of locks inside a “context lock”, because this may lead to a “deadlock” (e.g., A waiting for B that is waiting for C that is waiting for A => deadlock).

Hide   Copy Code

pContext-> m\_ContextLock.Lock();

*//… code code ..*

pContext2-> m\_ContextLock.Lock();

*// code code..*

pContext2-> m\_ContextLock.Unlock();

*// code code..*

pContext-> m\_ContextLock.Unlock();

The code above may cause a deadlock.

#### Rule of thumb #3:

Never access a client context outside the notification functions (e.g., Notify\*(ClientContext \*pContext)). If you do, you have to enclose it with m\_ContextMapLock.Lock(); … m\_ContextMapLock.Unlock();. See the source code below.

Hide   Copy Code

ClientContext\* pContext=NULL ;

m\_ContextMapLock.Lock();

pContext = FindClient(ClientID);

*// safe to access pContext, if it is not NULL*

*// and are Locked (Rule of thumbs#1:)*

*//code .. code..*

m\_ContextMapLock.Unlock();

*// Here pContext can suddenly disappear because of disconnect.*

*// do not access pContext members here.*

## 8 Future work

In future, the source code will be updated to have the following features in chronological order:

1. The implementation of AcceptEx(..) function to accept new connections will be added to the source code, to handle short lived connection bursts and DOS attacks.
2. The source code will be portable to other platforms as Win32, STL, and WTL.

## 9 F.A.Q

**Q1**: The amount of Memory used (server program is rising steadily on increase in client connections, as seen using the 'Windows Task Manager'. Even if clients disconnect, the amount of memory used does not decrease. What's the problem?

**A1**: The code tries to reuse the allocated buffers instead of releasing and reallocating it. You can change this by altering the parameters, iMaxNumberOfFreeBuffer and iMaxNumberOfFreeContext. Please review section 3.8.1.

**Q2**: I get compilation errors under .NET: "error C2446: '!=' : no conversion from 'unsigned int' to 'HANDLE'" etc.. What is the problem?

**A2**: This is because of the different header versions of the SDK. Just change the conversion to HANDLE so the compiler gets happy. You can also just remove the line #define TRANSFERFILEFUNCTIONALITY and try to compile.

**Q3**: Can the source code be used without MFC? Pure Win32 and in a service?

**A3**: The code was developed to be used with a GUI for a short time (not days or years). I developed this client/server solution for use with GUIs in an MFC environment. Of course, you can use it for normal server solutions. Many people have. Just remove the MFC specific stuff as CString, CPtrList etc.., and replace them with Win32 classes. I don’t like MFC either, so send me a copy when you change the code. Thanks.

**Q4**: Excellent work! Thank you for this. When will you implement AcceptEx(..) instead of the connection listener thread?

**A4**: As soon as the code is stable. It is quite stable right now, but I know that the combination of several I/O workers and several pending reads may cause some problems. I enjoy that you like my code. Please vote!

**Q5**: Why start several I/O workers? Is this necessary if you don’t have a true multiprocessor computer?

**A5**: No, it is not necessary to have several I/O workers. Just one thread can handle all the connections. On common home computers, one I/O worker gives the best performance. You do not need to worry about possible access violation threats either. But as computers are getting more powerful each day (e.g., hyperthreading, dual-core, etc.), why not have the possibility to have several threads? :=)

**Q6**: Why use several pending reads? What is it good for?

**A6**: That depends on the server development strategy that is adapted by the developer, namely “many concurrent connections” vs. “ high throughput server”. Having multiple pending reads increases the throughput of the server because the TCP/IP packages will be written directly into the passed buffer instead of to the TCP/IP stack (no double-buffering). If the server knows that clients send data in bursts, pending reads increase the performance (high throughput). However, every pending receive operation (with WSARecv()) that occurs forces the kernel to lock the receive buffers into the non-paged pool. This may lead to a WSAENOBUFFS error when the physical memory is full (many concurrent connections). The use of pending reads/writes have to be done carefully, and aspects such as “page size on the architecture” and “the amount of non-paged pool (1/4 of the physical memory)” have to be taken into consideration. Furthermore, if you have more than one IO worker, the order of packages is broken (because of the IOCP structure), and the extra work to maintain the order makes it unnecessary to have multiple pending reads. In this design, multiple pending reads is turned off when the number of I/O workers is greater than one because the implementation can not handle the reordering. (The sequence number must exist in the payload instead.)

**Q7**: In the previous article, you stated that we have to implement memory management using the VirtualAlloc function instead of new, why have you not implemented it?

**A7**: When you allocate memory with new, the memory is allocated in the virtual memory or the physical memory. Where the memory is allocated is unknown, the memory can be allocated between two pages. This means that we load too much memory into the physical memory when we access a certain data (if we use new). Furthermore, you do not know if the allocated memory is in physical memory or in virtual, and also you can not tell the system when "writing back" to hard disk is unnecessary (if we don’t care of the data in memory anymore). But be aware!! Any new allocation using VirtualAlloc\* will always be rounded up to 64 KB (page file size) boundary so that if you allocate a new VAS region bound to the physical memory, the OS will consume an amount of physical memory rounded up to the page size, and will consume the VAS of the process rounded up to 64 KB boundary. Using VirtualAlloc can be difficult: new and malloc use virualAlloc internally, but every time you allocate memory with new/delete, a lot of other computation is done, and you do not have the control to put your data (data related to each other) nicely inside the same page (without overlapping two pages). However, heaps are best for managing large numbers of small objects, and I shall change the source code so it only uses new/delete because of code cleanness. I have found that the performance gain is too small relative when compared to the complexity of the source code.

## 10 References

* “Developing a Truly Scalable Winsock Server using IO Completion Ports”, norm.NET, [Writing scalable server applications using IOCP](https://www.codeproject.com/KB/IP/iocp.aspx), 22/03/2005.
* “[Windows Sockets 2.0: Write Scalable Winsock Apps Using Completion Ports](http://msdn.microsoft.com/msdnmag/issues/1000/winsock/)”, Anthony Jones & Amol Deshpande, 22/02/2005.
* “A reusable, high performance, socket server class - Part 1-6”, Len Holgate, [A reusable, high performance, socket server class - Part 1](https://www.codeproject.com/KB/IP/jbsocketserver1.aspx), JetByte Limited, [jetbyte](http://www.jetbyte.com/).

Other sources

<https://codereview.stackexchange.com/questions/139420/tcp-server-using-winsock-io-completion-ports-model>

<http://www.drdobbs.com/cpp/multithreaded-asynchronous-io-io-comple/201202921?pgno=2>