SCALABLE CLIENT/SERVER SOLUTION USING WINDOWS OVERLAPPED I/O COMPLETION PORTS

Marin Lungu

Computer Dept., Faculty of Automation, Computers and Electronics
University of Craiova

Abstract: The paper will address the thorny issue of building a truly scalable client / server architecture using Windows Sockets. A comparative approach is taken towards the possible architectures for writing such a server, the limitations involved in each alternative are presented and the reasons behind choosing the Input Output Completion Ports (IOCP) solution are given. Some subtle details are pointed regarding the steps taken for tuning the server performance level by enhancing the Operating System's throughput. As an illustration, the solution presented implements the server architecture described and two separate messaging clients built for the Pocket PC platform and the Windows platform.

Keywords: Winsock Programming, IOCP, Client/Server Architectures, Communication networks.

1. INTRODUCTION

The solution presented is part of the trend of mobile applications development in today's IT market. It combines the power of a scalable highly computational server and the ease of use and flexibility advantage of mobile devices (like the Pocket PC platform the client uses).

For the purpose of this article some definition of the term "scalability" is necessary. In terms of programming, scalability means the ability of an application to fully utilize the additional processing power available on a multiprocessor system. Also for an application to be considered scalable it should be able to service a large number of clients. Putting these two definitions together leads to a new term that is often used: "scaling up" an application.

The presented architecture is such an example. The choice of it emerged from a theoretical analysis of the alternative architectures for building a high-load server, (Richter *et al*., 1999); as well as from comparative performance reports. The solution's goal was to build a scalable client server application for mobile-to-mobile or mobile-to-desktop messaging. A custom message-oriented communication protocol was developed. As such the focus was on designing a solution optimized to scale well to a high number of simultaneous connections (thousands) rather than for heavy-load data transfer.

The article will argument why I/O Completion Port based server design emerged as a natural choice, it will explain the multithreading model under the hood and the limitations that it overcame. Also there will be a description of the implementation of the fine tuning segments for boosting up the Operating System's performance.

1.1. Analyzing the server model architectures in Winsock Programming

Windows sockets perform I/O operations in 2 operating modes: *blocking* and *non-blocking*. The blocking mode implies that function calls involving I/O operations such as *send* and *recv* wait until the operation completes before returning control to the application. In the non-blocking mode, control returns immediately even if the function could not send or receive data. For each mode, different models are available.

The blocking model. This is the simplest model from the development point of view. Usually, applications implementing this method function in a consumer/producer manner. As far as the receiving operation is concerned, one thread (the producer) is provided for receiving the data from network and storing it in a thread shared queue, while another (the consumer) solves the application computation of data by getting it from the queue. The same applies for the send operation. The disadvantage of this model of operation is that it does not scale well once it starts dealing with a large number of live sockets.

The described consumer/producer model could be extended to implement a consumer and a producer thread per connected socket. Allocating a thread per accepted connection has certain weak points. Apart from the high level of housekeeping, from a certain number of threads up, the so called "Thread Thrashing" comes into play in the Operating System. In the usual workflow, each running thread is allocated a certain quantum of time to run. If it does not finish in the provided time, the thread is set on hold while the OS changes the context for the next thread to run.

Although this mode has a plus in the simplicity of use, on heavy load the number of threads required is high; the system's performance drops drastically since most of its execution time goes into context switching rather than in the active processing. If the number of allocated threads is limited, connections are refused.

The non-blocking models. Four non-blocking models have been analyzed in order to determine the best choice: *WSAAsyncSelect*, *WSAEventSelect*, *overlapped* and *completion port*. A short description of each advantages and drawbacks will follow. (Quinn *et al*., 1995);

WSAAsyncSelect: Windows provides an asynchronous I/O model that allows an application to receive Windows message-based notifications regarding network events on a socket. This functionality is available by calling the *WSAAsyncSelect* function after creating the socket. (Jones *et al*., 2002)

When a network event occurs (such as incoming data becoming available for reading or buffer space for writing) an event is posted for the window's message pomp. No asynchronous data transport is available, only notification on occurring events.

To make use of this model the application has to initialize a window and define a custom message that identifies a certain event occurring for a given socket. The prototype for this function is the following:

```
WSAAsyncSelect(s, hwnd, WM_SOCKET,
  FD_CONNECT|FD_READ|FD_WRITE| FD_CLOSE);
```

The first parameter *s* identifies the socket, *hwnd* is the receiving window for the message, *WM_SOCKET* is the ID of the message and the rest are flags set for the events we are interested in marking (*connect, read* possible, *write* possible, *close*). After a successful call to *WSAAsyncSelect*, the designated window starts receiving windows messages when an event it is interested in occurs. The plus for this architecture is that it allows the management of multiple connections without much programming overhead. As drawbacks, it enforces the use of a window even if the application itself would not require it and, most important it induces a performance bottleneck. Since a window services all the incoming notification messages, it is clear that the system's performance is limited to a number of simultaneous connections and the solution is not scalable.

WSAEventSelect. This model is similar to the previous model in that is delivers a means for an application to be notified of the same events occurring on a socket as the previous function. Instead of sending a windows-message, it signals the appearance of an event the application is interested in by setting an event. The application's working thread waits in an endless loop for this event to be set and treats it. The "catching" function is:

```
DWORD WSAWaitForMultipleEvents(
   DWORD cEvents,
   const WSAEVENT FAR * lphEvents,
   BOOL fWaitAll,
   DWORD dwTimeout,
   BOOL fAlertable
);
```

The *cEvents* and *lphEvents* parameters identify an array of events on which the function is waiting (*cEvents* – the number of elements, *lphEvents* – pointer to the array of events). Although this model allows multiple simultaneous connections treated asynchronously, it has a serious limitation in the maximum number of events the *WSAWaitForMultipleEvents* can wait on. This is defined as *WSA_MAXIMUM_WAIT_EVENTS* and is 64, so we can have at most 64 simultaneous connections.

The Overlapped Model. It offers applications better system performance that any of the above models. The main difference is that not only asynchronous notification is available upon network events occurrence but also data transfer. The overlapped model's allows an application to post one or more asynchronous I/O requests at a time using an additional overlapped parameter. This parameter contains a buffer pointer and the size of it. When using the overlapped socket model the TCP stack is instructed to directly perform I/O using the buffer provided in the function call. Therefore in addition to the non-blocking advantage of the overlapped sockets there is an increase of performance because a buffer copy is saved between the TCP stack and the user buffer for each call.

Once an overlapped I/O request finalizes, the application is responsible for retrieving the overlapped results. An event handler can be submitted as a parameter to the call to the I/O request. In this case the Operating System will set that event when the operation completes. However, although improved in performance from the *WSAEventSelect* by the overlapped buffer used directly in the date transfer, if the event notification is used there persists the WSA_MAXIMUM_WAIT_EVENTS limitation of maximum 64 simultaneous connections. So this model alone does not scale very well.

The Completion Port model. Completion ports are the mechanism by which an application uses a pool of threads to service the completion of I/O operations. These threads are created when the application is started. The Completion Port is in fact a thread safe queue that the Operating System makes available to the application and which it uses to notify the application that some I/O request was completed. (Holgate L. ,2002).
The Completion Port is initialized by calling the *CreateIOCompletionPort* function that associates an I/O completion port with a bind socket. The most important parameter passed to this function is its concurrency value that is specified when the CP is created. This value limits the number of threads associated with the completion port that can run simultaneously. When a packet is queued to a CP, the OS first checks if the number of currently running threads is bellow the concurrency limit. If so, a new thread is started to process the completion packet, otherwise it waits until the number of running threads goes below this limit.
The running threads wait for a packet on the Completion Port by calling the following function:

```
BOOL GetQueuedCompletionStatus(
  HANDLE CompletionPort,
 LPDWORD lpNumberOfBytes,
 PULONG_PTR lpCompletionKey,
LPOVERLAPPED* lpOverlapped,
 DWORD dwMilliseconds
);
```

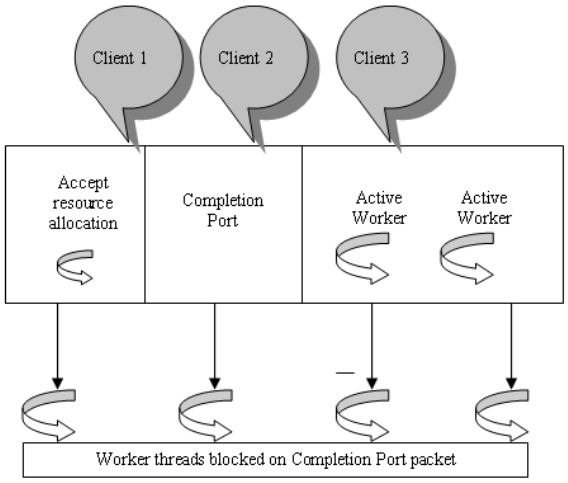

Fig. 1 I/O Overlapped Completion Ports Architecture

The function specifies the handle of the Completion Port to wait on, receives the number of bytes transferred, a pointer to the overlapped structure passed on the I/O request call, time in ms to wait before returning and some additional I/O information.
This system overcomes the limitation of the simultaneous connection present in the last presented models. By analyzing the performance increase of the Overlapped sockets (eliminating the extra copy of the data in the user's buffer) and the advantages of the Completion Ports we reached the proposed architecture for the scalable server : I/O Overlapped Completion Port.

1.2. Special considerations for the Overlapped I/O Completion Port implemented architecture

Once a completion port is created socket handles are associated with it. The next step is to create the worker threads to service the Completion Port. The number of such worker threads is an issue. This is where the scalability can be enhanced. There is a difference between the worker threads created and the number of threads that the Completion Port is instructed to allow to run simultaneously.
The ideal case would be to always have one thread running for each processor in order to prevent context switching and Operating System's cache miss. However, if the number of available threads equals the number of processors performance may drop in the case one of these threads go blocked.

```
GetSystemInfo(&m_SystemInfo);
for(DWORD i = 0;
 i < m_SystemInfo.dwNumberOfProcessors*2; i++)
{
      HANDLE ThreadHandle;
     // Create a server worker thread, and pass the
     // completion port to the thread.
     ThreadHandle = CreateThread(NULL, 0,
       ServerWorkerThread,m_hCompletionPort,0,NULL);

          IncWorkerCount();
          // Close the thread handle
          CloseHandle(ThreadHandle);
}
```

In this case, even if the concurrency limit of the Completion Port allows for another thread to be awaken, there is none available. A rule of thumb is to create 2* Number_Of_Processors worker threads (as in the above code section).

Due to the nature of the application (heavy load on number of simultaneous connections) the server is optimized to scale to this. A separate thread is used for accepting the connections. The function used is *AcceptEx,* one of the Microsoft extension functions, the only Winsock API capable of using overlapped I/O to accept connections on a socket. In a normal synchronous accept function call, the new socket is a return parameter of the API call. However, since the

creation of a socket is the most time consuming operation, it is desired to create the socket in advance and have it in a "ready" state for any incoming connections. This is why *AcceptEx* requires an already created socket as a parameter. In order for this algorithm to be effective, there has to be maintained a pool of already created sockets. The number of these needs not be too high because there would be too much server resources wasted (all the connected sockets are allocated from the non-paged pool). It should not be too small either otherwise incoming connections would have to wait for new sockets to be created.

The algorithm implemented is to have two separate lists one with "ready sockets" and another with "active sockets". An initial number of sockets is created in the beginning of the application.
When a connection is completed, the handle is passed from the "ready sockets" to the "active sockets". A limit to the number of created sockets was imposed in order not to overload the system above a certain level.

Another feature of *AcceptEx* is that the function can receive data and accept a client connection in a single step. That is the function does not return until at least a byte of data was received. This increase in performance was well suited to the messaging architecture that was built where the protocol imposes that a message is sent on each connection.

What is a significant benefit can also prove to be a fatal flaw if a malicious client is on the other end of the line. When using *AcceptEx* it is possible for a client to overload the server and eventually block it (from lack of resources in the non-paged poll) if it initiates connections without sending data. There is no other way to prevent this but to periodically check the connections in the "active sockets" for a timeout. We need to determine which of the "active" sockets is pending, waiting for data and for how long they have been waiting. This is done by calling the *getsockopt()* function with the SO_CONNECT_TIME that returns -1 if the socket is not connected or the time since the socket is pending.

CLIENT/SERVER IMPLEMENTATION DETAILS

The client being implemented on Pocket PC does not use the Overlapped Completion Port method. One reason is that it is not necessary for the client to reach a performance level comparable to the server and the other that that model is not available for Windows CE. The client has a single worker thread that periodically checks if the user has issued any new message, takes that message (in a thread safe critical section) and delivers it to the server. The server checks if there are any messages for the client and sends them in a row.

Due to the multithreading programming approach, the clean shutdown both for the client and server required special handling. Upon shutdown, the server first closes all connected sockets and then the worker threads by placing a special KillPacket (some zero and NULL values present on the packet) on the Completion Port that the worker threads catch and shut down. There are posted on the Completion Port as many KillPacket as events created in the beginning of the application.

```
for (long li = 0; li< m_ldWorkerThreads; li++){
  PostQueuedCompletionStatus(m_hCompletionPort ,0 ,0,
   NULL);
  DecWorkerCount();
  Sleep(500);
 }
```

The Server worker thread waits in an endless loop and intercepts all posted packets. If it identifies the data for the KillPacket, it exits.

```
BOOL ret = GetQueuedCompletionStatus(CompletionPort,
        &BytesTransferred,(LPDWORD)&pPerHandleData,
        &lpOverlapped, INFINITE);
   if((BytesTransferred==0)&&(pPerHandleData==0)){
            DeleteCriticalSection(&CS);
            return 0;
     }
```

Else it receives the data transmitted.

```
if (pPerIoData->OperationType == RECV_POSTED)
         CString sTemp(pPerIoData-
>DataBuf.buf,pPerIoData->DataBuf.len);
```

CONCLUSION

Writing a network application is not difficult, as was presented on the simpler Winsock I/O models; however writing a scalable application is challenging. With the Overlapped I/O Completion Port model and some extension Microsoft functionalities major bottlenecks can be overcome.

REFERENCES

Holgate, Len (2002).A reusable, high performance, socket server class; in *(http://www.codeproject.com/internet/JBSocketServer3.asp*)

Jones, A. and J. Ohlund; (2002); *Network Programming for Microsoft Windows;* Microsoft; 2 edition, ISBN-13: 978-0735615793

Roberts, D. (1995); *Developing for the Internet with WinSock;* Publisher: Coriolis Group Books; ISBN-13: 978-1883577421

Richter, J., J. D. Clark (1999); *Programming Server-Side Applications for Microsoft Windows 2000;* Microsoft Press; ISBN-13: 978-1572319967

Solomon, D. A., M. Russinovich (2000); *Inside Microsoft Windows 2000*; Microsoft Programming Series.

Quinn B., D. Shute, D. K. Shute; (1995); *Windows Sockets Network Programming*; Addison-Wesley Advanced Windows Series; ISBN-13: 978-0201633726