Networking

Network access is a fundamental capability in most modern applications. Server applications that process client requests strive to maximize scalability and their throughput capacity to serve clients faster and to serve more clients per server, whereas clients aim to minimize network access latencies or to mitigate its effects. This section provides advice and tips for maximizing networking performance.

Network Protocols

The way an applicative network protocol (OSI Layer 7) is constructed can have a profound influence on performance. This section explores some optimization techniques to better utilize the available network capacity and to minimize overhead.

Pipelining

In a non-pipelined protocol, a client sends a request to server and then waits for the response to arrive before it can send the next request. With such a protocol, network capacity is under-utilized, because, during the network round-trip time (i.e. time it takes network packets to reach the server and back), the network is idle. Conversely, in a pipelined connection, the client can continue to send more requests, even before the server has processed the previous requests. Better yet, the server can decide to respond to requests out of order, responding to trivial requests first, while deferring processing of more computationally-demanding requests.

Pipelining is increasingly important, because, while Internet bandwidth continues to increase worldwide, latency improves at a much slower rate because it is capped to physical limits imposed by the speed of light.

An example of pipelining in a real-world protocol is HTTP 1.1, but it is often disabled by default on most servers and web browsers because of compatibility issues. Google SPDY, an experimental HTTP-like protocol supported by Chrome and Firefox web browsers, as well as some HTTP servers, and the upcoming HTTP 2.0 protocol mandate pipelining support.

Streaming

Streaming is not just for video and audio, but can be used for messaging. With streaming, the application begins sending data over the network even before it is complete. Streaming reduces latency and improves network channel utilization.

For example, if a server application fetches data from a database in response to a request, it can either read it in one chunk into a `DataSet` (which can consume large amounts of memory) or it can use a `DataReader` to retrieve records one at a time. In the former approach, the server has to wait until the entire dataset has arrived before it can begin sending a response to the client, whereas in the latter approach, the server can begin sending a response to the client as soon as the first DB record arrives.

Message Chunking

Sending small chunks of data at a time over the network is wasteful. Ethernet, IP, and TCP/UDP headers are no smaller, because the payload is smaller, so although bandwidth utilization can still be high, you end up wasting it on headers, not actual data. Additionally, Windows itself has per-call overhead that is independent or weakly dependent on data chunk size. A protocol can mitigate this by allowing several requests to be combined. For example, the Domain Name Service (DNS) protocol allows a client to resolve multiple domain names in one request.

Chatty Protocols

Sometimes, the client cannot pipeline requests, even if the protocol allows this, because the next requests depend on the content of the previous replies.

Consider an example of a chatty protocol session. When you browse to a web page, the browser connects to the web server via TCP, sends an HTTP GET request for the URL you are trying to visit, and receives an HTML page as a response. The browser then parses the HTML to figure out which JavaScript, CSS, and image resources it has to retrieve and downloads them individually. The JavaScript script is then executed, and it can fetch further content. In summary, the client does not immediately know all the content it has to retrieve to render the page. Instead, it has to iteratively fetch content until it discovers and downloads all content.

To mitigate this problem, the server might hint to the client which URLs it will need to retrieve to render the page and might even send content without the client requesting it.

Message Encoding and Redundancy

Network bandwidth is often a constrained resource, and having a wasteful messaging format does not help performance. Here are a few tips to optimize messaging format:

- Do not transmit the same thing over and over, and keep headers small.
- Use smart encoding or representation for data. For example, strings can be encoded in UTF-8, instead of UTF-16. A binary protocol can be many times more compact than a human-readable protocol. If possible, avoid encapsulations, such as Base64 encoding.
- Use compression for highly compressible data, such as text. Avoid it for uncompressible data, such as already-compressed video, images, and audio.

Network Sockets

The sockets API is the standard way for applications to work with network protocols, such as TCP and UDP. Originally, the sockets API was introduced in the BSD UNIX operating system and since became standard in virtually all operating systems, sometimes with proprietary extensions, such as Microsoft's WinSock. There are a number of ways for doing socket I/O in Windows: blocking, non-blocking with polling, and asynchronous. Using the right I/O model and socket parameters enables higher throughput, lower latency, and better scalability. This section outlines performance optimization pertaining to Windows Sockets.

Asynchronous Sockets

.NET supports asy nchronous I/O through the `Socket` class. Howev er, there are two f amilies of asy nchronous APIs: `BeginXXX` and `XXXAsync`, where XXX stands f or Accept, Connect, Receiv e, Send, and other operations. The f ormer use the .NET Thread Pool's registered wait capability to await ov erlapped I/O completion, whereas the latter use .NET Thread Pool's I/O Completion Port mechanism, which is more perf ormant and scalable. The latter APIs were introduced in .NET Framework 2.0 SP1.

Socket Buffers

Socket objects expose two settable buf f er sizes: `ReceiveBufferSize` and `SendBufferSize`, which specif y buf f er sizes allocated by the TCP/IP stack (in OS memory space). Both are set to 8,192 by tes by def ault. The receiv e buf f er is used to hold receiv ed data not y et read by the application. The send buf f er is used to hold data that has been sent by the application but that has not y et been acknowledged by the receiv er. Should the need to retransmit arise, data f rom the send buf f er is retransmitted.

When the application reads f rom the socket, it unf ills the receiv e buf f er by the amount read. When the receiv e buf f er becomes empty , the call either blocks or becomes pending, depending if sy nchronous or asy nchronous I/O is used.

When the application writes to the socket, it can write data without blocking until the send buf f er is f ull and cannot accommodate the data or until the receiv er's receiv e buf f er becomes f ull. The receiv er adv ertises how f ull its receiv e buf f er size is with each acknowledgement.

For high-bandwidth, high latency connections, such as satellite links, the def ault buf f er sizes may be too small. The sending end quickly f ills up its send buf f er and has to wait f or acknowledgement, which is slow to arriv e because of high latency . While waiting, the pipe is not kept f ull and the endpoints only utilize a f raction of the av ailable bandwidth.

In a perf ectly reliable network, the ideal buf f er size is the product of the bandwidth and latency . For example, in a 100Mbps connection with a 5 ms round-trip time, an ideal buf f er window size would be (100,000,000 / 8) × 0.005 = 62,500 by tes. Packet loss reduces this v alue.

Nagle's Algorithm

As mentioned bef ore, small packets are wastef ul, because packet headers may be large compared to the pay load. Nagle's algorithm improv es perf ormance of TCP sockets by coalescing multiple writes by the application into up to a whole packet's worth of data. Howev er, this serv ice does not come f or f ree as it introduces delay bef ore data is sent. A latency sensitiv e application should disable Nagle's algorithm by setting the `Socket.NoDelay` property to true. A well-written application would send large buf f ers at a time and would not benef it f rom Nagle's algorithm.

Registered I/O

Registered I/O (RIO) is a new extension of WinSock av ailable in Windows Serv er 2012 that prov ides a v ery ef f icient buf f er registration and notif ication mechanism. RIO eliminates the most signif icant inef f iciencies in Windows I/O:

- User buf f er probing (checking f or page access permissions), locking and unlocking (ensuring the buf f ers are resident in RAM).
- Handle lookups (translating a Win32 `HANDLE` to a kernel object pointer).
- Sy stem calls made (e.g. to dequeue I/O completion notif ications).

These are "taxes" paid to isolate the application f rom the operating sy stem and f rom other applications, to the end of ensuring security and reliability . Without RIO, y ou pay these taxes per call, which, at high I/O rates, becomes signif icant. Conv ersely , with RIO, y ou incur the costs of the "taxes" only once, during initialization.

RIO requires registration of buf f ers, which locks them in phy sical memory until they are de-registered (when the application or subsy stem uninitializes). Since the buf f ers remain allocated and resident in memory , Windows can skip probing, locking, and unlocking per call.

RIO request and completion queues reside in the process' memory space and are accessible to it, meaning a sy stem call is no longer required to poll the queue or dequeue completion notif ications.

RIO supports three notif ication mechanisms:

- Polling: This has the lowest latency but means a logical processor is dedicated to polling the network buf f ers.
- I/O Completion Ports.
- Signaling a Windows ev ent.

At the time of writing, RIO was not exposed by the .NET Framework, but it is accessible through the standard .NET interoperability mechanisms (discussed in Chapter 8).