Apache 2.0 for Windows

Bill Stoddard
Apache Software Foundation
Senior Software Engineer, IBM Corp.
stoddard@apache.org
October 23, 2000

Introduction 1
Apache 2.0 Major Architectural Features 2
  The Apache MPM 2
  Apache Portable Runtime 3
  Module Hook API 3
  Filter API 4
Apache 2.0 for Windows: Specializing for Win32 4
  mpm_winnt 4
  Advanced Windows APIs used by mpm_winnt and APR 5
    AcceptEx 5
    TransmitFile 6
  Completion Ports 6
    CreateIoCompletionPort() 6
    GetQueuedCompletionCompletionStatus 7
    Completion Ports and Over Scheduling 7
  mod_file_cache 8
Other Enhancements in Apache 2.0 for Windows 8
Apache 2.0 for Windows Performance Numbers 10
Future Enhancements 10
Where to find this write-up and presentation 10

Introduction

Apache 1.3.0 was the first release of Apache that included support for the Windows operating system (NT & 95). The Apache Server was designed for Unix like operating systems by Unix developers. The Windows port illustrates the difficulties of porting a 'classic' Unix application to Windows.

The first major obstacle to overcome was the difference in request dispatch model. Apache on Unix dispatches a process-per-request. While this is an acceptable model for a classic Unix application, the process-per-request model is non-optimal for a Windows application as Windows processes are quite heavy relative to Unix processes. Windows applications typically use threads to do the work processes do in classic Unix applications. Apache 1.3 and Apache 2.0 for Windows implement a thread-per-request dispatch model, so none of the process/thread control code could be shared between Unix and Windows. This contributed to a rat's nest of code in Apache 1.3's http_main.c.

The second major obstacle is the lack of a fork() system call on Windows. Apache for Unix relies heavily on the property of the fork() system call to create a near exact replica of the calling (or parent) process. A running instance of Apache on Unix

consists of a parent process (the first process started) and one or more child processes. The child processes are responsible for handling the requests. The parent is responsible for reading and parsing the server configuration and storing it in memory, for opening the listening sockets, and for opening the server logs (access and error logs). The parent never sees requests. Once the parent does all its work, it fork()'s multiple child processes and each forked process has a near exact replica of the parent processes address space, including open socket descriptors, memory, etc. Thus, the child processes are saved the overhead of reading the server configuration file, opening logs, etc. Lack of fork() on Windows is another reason Windows could not efficiently implement a process-per-request dispatch model. Each process (and there could be 1000's of them) would have to suffer start-up overhead.

Overcoming these two obstacles required some major hacks to the server and once the port was completed, the Windows server was ½ as fast as Apache on Linux running on identical Intel hardware. This pointed out the third major difficulty that was never addressed in the original port to Windows: performance.

We have learned that to get optimal performance out of a Windows server requires use of advanced Windows APIs. Apache 2.0 for Windows is up to 2.5 X faster than Apache 1.3 serving static pages

Apache 2.0 Major Architectural Features

The Apache MPM

Dean Gaudet is credited with the breakthrough notion of dedicating an Apache module to the task of dispatching requests to an execution context. This special purpose module is called a Multi-Processing Module, or MPM. An MPM is responsible for starting and managing threads and/or processes, accepting connections off TCP's so_q and dispatching these connections to a thread or process. There is no protocol specific code implemented in an MPM.

The MPM architecture enables developers to write MPMs that leverage platform specific features while keeping this code out of the core server. At the time of this writing there are at least 8 MPMs available:

- PREFORK
  The prefork MPM implements the traditional Apache process-per-request dispatch model. There is one parent process and one or more single threaded child processes for handling requests. As in Apache 1.3 for Unix, the number of child processes is allowed to vary with server load. This MPM offers superior robustness: if one child process dies, only the client connected to that process is affected.

- MPMT-PTHREAD
  The mpmt-pthread MPM implements a thread-per-request dispatch model. There is a single parent process and one or more multithreaded child processes. The number of child processes is allowed to vary with server load. The number of threads per child process is fixed.

- DEXTER
  The dexter MPM is similar to the mpmt-pthread MPM except that the number of child processes is fixed while the number of threads is allowed to vary with server load.

- MPM_WINNT
  The winnt MPM implements a thread-per-request dispatch model. There is a single parent and a single child with a fixed number of threads.

- OS2

- BEOS

- PERCHILD
  This is an experimental MPM designed to dispatch requests to backend server processes each running under a pre-configured user/group.

- STM
  This is an experimental high performance MPM created by Mike Abbott of SGI that implements a non-preemptive thread-per-request dispatch model. This MPM, which currently only runs on SGI's OS, is not currently part of the Apache distribution.

Apache Portable Runtime

The Apache Portable Runtime (APR) implements a platform independent API for Accessing common operating system services. The goal of APR is to enable application developers to write code that can be seamlessly shared across multiple operating systems while still providing access to the 'best' OS specific APIs for implementation. For example, Windows implements a BSD socket API, which is compatible with Unix. A application written to BSD sockets will not be able to cleanly access Win32 specific APIs like TransmitFile(), AcceptEx(), WSARecv(), et. al.

APR currently implements APIs in the following categories:

- File and pipe I/O
- Network I/O
- Process & thread management
- Memory management, including the Apache pool API
- Time keeping and conversion
- Common error status management

Module Hook API

The module hook API replaces Apache 1.3's static module hook structure. In Apache 2.0, modules call ap_hook_*() functions to register their hooks, where * is the specific hook. E.g. ap_hook_pre_config(), ap_hook_post_config(), etc. The essential benefit of the new API to module writers is that adding a new hook to the core server does

not require source code changes to existing modules that are not interested in the new hook.

Filter API

Conceptually, the filter API enables the output of handlers to be chained to other module handlers. E.g., a CGI can generate content marked up with SSI tags and that marked up content can be directed through mod_include's SSI filter. The details are significantly more complex. The filter API is still in early stages of development and will likely change significantly over the next few releases of Apache.

Apache 2.0 for Windows: Specializing for Win32

Apache 2.0 uses advanced Windows APIs (in both mpmt_winnt MPM and APR) and a file handle cache (mod_file_cache.c) to serve static pages at up to 150% faster than Apache 1.3. This section will review the specific enhancements made to Apache to achieve this performance boost. Mpmt_winnt will also run under Windows 95/98 (with some debugging) but the performance improvements will only be available under Windows NT/2000.

mpm_winnt

Th mpm_winnt MPM implements a thread-per-request dispatch model. The parent process (single threaded) starts exactly one multithreaded child process. The number of threads in the child is set by a configuration directive (ThreadsPerChild) and does not change after the server is started.

When the server is started, the parent process performs the following steps (some steps are omitted for brevity. Read the code for all the glorious details):

- *Opens and read httpd.conf*
- *Opens log files*
- *Create AcceptEx IOCompletion port*
- *Open listen sockets*
- *Associate listeners with the completion port*
- *Create child process*
- *Duplicate listen sockets and send them to the child process*
- *Duplicate the AcceptEx completion port and send it to the child process*
- *Wait for child death, shutdown or restart event*

The main thread in the child process perform the following steps:

- *Opens and reads httpd.conf*
- *Opens logs files*
- *Reads listen sockets off of pipe from parent*
- *Reads AcceptExComplationPort off of pipe from parent*
- *Starts ThreadsPerChild worker threads*
- *Creates n AcceptEx completion contexts for each listening socket and calls AcceptEx asynchronously for each context*

- *Blocks waiting for exit_event (signaled by parent in response to shutdown or restart event) or a maintenance_event signaled by one of the worker threads*

Each worker thread in the child process perform the following steps:

- *Conditionally resets the AcceptEx completion context if this thread has just handled a connection.*
- *Blocks on GetQueuedCompletionStatus() waiting for asynchronous i/o completion packets*
- *When GetQueuedCompletionStatus( ) unblocks, if more AcceptEx completion contexts need to be created, signal a maintenance event*
- *Recover the completion context for the accepted connection*
- *Handle the accepted connection*

The AcceptEx completion context saves state information across the asynchronous AcceptEx call.

```
typedef struct CompContext {
   OVERLAPPED Overlapped;
   SOCKET accept_socket;
   apr_socket_t *sock;
   ap_listen_rec *lr;
   BUFF *conn_io;
   char *recv_buf;
   int  recv_buf_size;
   apr_pool_t *ptrans;
   struct sockaddr *sa_server;
   int sa_server_len;
   struct sockaddr *sa_client;
   int sa_client_len;
} COMP_CONTEXT, *PCOMP_CONTEXT;
```

Advanced Windows APIs used by mpm_winnt and APR

AcceptEx

AcceptEx is a Microsoft specific extension to the Winsock API. This call is not currently implemented in any APR functions. AcceptEx is used exclusively by the mpm_winnt MPM.

```
BOOL AcceptEx(
  SOCKET sListenSocket,
  SOCKET sAcceptSocket,
  PVOID lpOutputBuffer,
  DWORD dwReceiveDataLength,
  DWORD dwLocalAddressLength,
  DWORD dwRemoteAddressLength,
  LPDWORD lpdwBytesReceived,
  LPOVERLAPPED lpOverlapped
);
```

The notable features of AcceptEx are:

- Can be called in asynchronous mode (i.e., call returns immediately, similar to non-blocking I/O)
- Calling AcceptEx() asynchronously enables dispatching worker threads off of a completion port via a call to GetQueuedCompletionStatus()
- The accept socket is explicitly passed in, which enables reusing the accept socket in certain situations
- It can be made to receive the first data packet after accepting the connection (a.k.a., Accept and Receive)
- It returns the local/remote addresses (saves a system call to get them when using accept())

TransmitFile

TransmitFile is a Microsoft specific extension to the Winsock API. The apr_sendfile() call is implemented with TransmitFile().

```
BOOL TransmitFile(
  SOCKET hSocket,
  HANDLE hFile,
  DWORD nNumberOfBytesToWrite,
  DWORD nNumberOfBytesPerSend,
  LPOVERLAPPED lpOverlapped,
  LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
  DWORD dwFlags
);
```

The notable features of the TransmitFile API are:

- Eliminates multiple buffer copies and system calls by aggregating read()/send()/closesocket()
- If you can reliably detect if this is the last request on the connection, dwFlags can be set to cause TransmitFile to initiate a disconnect, which will allow the accept socket to be reused on the AcceptEx() call.
- The file handle must have been opened using the Windows specific CreateFile() API.
- It can send chunks of header/trailer data along with the file (particularly useful for sending HTTP headers along with the file)
- Tight integration with the OS file cache

Completion Ports

Windows completion ports are one of several objects in Windows that can be used to receive notification that an asynchronous I/O event has completed. They are remarkably useful for implementing high performance network servers.

CreateIoCompletionPort()

A completion port is created with the CreateIoCompletionPort() call.

```
HANDLE CreateIoCompletionPort (
  HANDLE FileHandle,               // handle to file
  HANDLE ExistingCompletionPort,   // handle to I/O completion port
  ULONG_PTR CompletionKey,         // completion key
  DWORD NumberOfConcurrentThreads // number of threads to execute concurrently
);
```

FileHandle can also be a SOCKET that is enabled for async I/O. Sockets created with the sockets() call are, by default, opened in blocking mode but 'enable' for async I/O.

In mpm_winnt, the parent creates the completion port, thusly:

```
AcceptExCompPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE,
NULL, 0, 0);
```

A listen socket is associated with the completion port thusly:

```
CreateIoCompletionPort(AcceptExCompPort, listen_socket, 0, 0);
```

This code causes a completion port object to be created in the NT kernel. When any asynchronous socket operations issued on the listening socket completes, NT will queue an I/O completion packet to the port. The queuing of an I/O completion packet causes a thread blocked on GetQueuedCompletionStatus to be awakened to handle the completed async socket I/O. The I/O completion packet contains enough information (the completion key) to allow the thread to regain context (what async call is completing, what to do when the call completes, pointers to relevant storage, etc. See COMP_CONTEXT above.) in order to correctly handle the request.

Notice the *NumberOfConcurrentThreads* argument. The story behind this argument is quite interesting! See the section titled "Overscheduling" for more info.

GetQueuedCompletionCompletionStatus

All the worker threads first issue an asynchronous AcceptEx() then block on this call awaiting notification (in the form of I/O completion packets) that a connection has been received. In general, notification can flow to the completion port for any async I/O issued against socket, not just AcceptEx().

```
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
);
```

A system that has too many 'active' threads can suffer performance degradation due to 'over scheduling'. In other words, having too many active threads can cause excessive time to be spent by the OS managing thread context switches, time that would better be spent doing productive work. Over scheduling is a major performance problem with Apache 1.3's process-per-request model. The processes are dispatched more or less at random to handle requests. This becomes a serious problem as the number of concurrent clients (number of processes) increases.

Completion ports provide an interesting feature that can help prevent over scheduling. A completion port can be instructed (at creation) to only allow a fixed number of concurrently active threads (*NumberOfConcurrentThreads*). A value of 0 means the completion port will allow one active thread per CPU in the system. An active thread is defined as a thread that is not blocked on I/O. If an active thread blocks on I/O, the

completion port will allow another thread to be dispatched to handle work pending on the port (in our case, accepted connections).

Another interesting feature of completion ports is that threads blocked on GetQueuedCompletionStatus() are dispatched in LIFO order. The most recently active thread is given the next piece of work to do. If a thread calls GetQueuedCompletionPort() and there is work available, that thread will get the work rather than blocking. This pretty much ensures that the thread's stack will still be paged in, etc, which is good for performance.

An ideal server (from a performance perspective) would set the number of active threads to the number of CPUs in the system and never block on I/O. In practice, some threads get caught up in CPU intensive activities, so *NumberOfConcurrentThread*s should be set to something greater than 0 but less than *ThreadsPerChild.* We are still experimenting with the correct value in Apache 2.0 for Windows. Furthermore, Apache 2.0 for Windows will block on network I/O. The most common case is blocking on a persistent connection read(). Even with these limitations, Apache 2.0 for Windows is significantly faster serving static pages than Apache 1.3 and should be reasonably competitive with Apache on Linux and MS IIS in terms of raw performance and scalability.

mod_file_cache

Stat'ing, opening and closing files are very expensive operations on Windows. Eliminating this overhead by caching open file handles results in a substantial performance improvement when serving static files.

mod_file_cache is an extension of Dean Gaudet's mod_mmap_static. It can be used exactly like mod_mmap_static on Unix platforms (same configuration directives, etc.). For platforms that support apr_sendfile() (Windows NT/2000 and some Unix platforms), mod_file_cache can be directed to cache open file handles instead of mmap'ed files.

Caching file handles offers several advantages over caching MMAP'ed files. First, the storage used by a file handle cache entry is a few bytes as compared to the entire file for an MMAP'ed cache entry. Second, caching the file handle offloads memory management responsibility to the file system. An infrequently hit file in the handle cache will be paged out, freeing up system resources. Most importantly, offloading the memory management to the file system allows the cache to be dynamically loaded at runtime (and eliminating a configuration step) without much danger of consuming excessive system resources. Mod_file_cache does not implement dynamic cache loading at this time.

Caching file handles also introduces some difficulties. The most notable is that you should not cache file handles for content that is to be served over SSL sessions.

Other Enhancements in Apache 2.0 for Windows

There have been a number of other noteworthy enhancements to Apache 2.0 for Windows by B. Stoddard, W. Rowe, T. Costello, G. Marr, et. al. worth mentioning. See the code and CHANGES file for more details.

- CGI scripts running on Windows NT/2000 are now able to flush partial responses to the network. Apache on Unix has always had this capability
- Code to manage starting stopping and running Apache as a service has been substantially improved.
- Mod_isapi is significantly improved.
- Eliminate DLL relocation at start-up. This will help with debugging segmentation faults in the field by allowing us to precisely locate the offending line of code.
- Enhance the make/build environment. There is now a single, all encompassing apache.dsw project file that can be used to compile Apache directly or it can be used to export the project makeefiles.
- Other enhancements…

Apache 2.0 for Windows Performance Numbers

Table 1 – HTTP/1.0 Connections per second w/o keep-alive

File Size (bytes)	Apache 1.3	Apache 2.0 AeTf	Apache 2.0 Cache	Apache 2.0 socket reuse	Apache 2.0 AeTf, cache, socket reuse
500	419	470	763	592	1088
1000	416	468	750	585	1065
4000	375	450	726	556	1004
8000	340	435	664	540	918
16000	294	396	610	484	696@85% cpu
20000	261	377@98%			
32000	219				
64000	147	173@55%	172@40%	178@54%	178@38%

Future Enhancements

The primary performance enhancements planned for Apache for Windows is to handle all network I/O asynchronously. This will enable supporting 10s of thousands of concurrent HTTP/1.1 keep-alive clients. When all network I/O can be guaranteed to be asynchronous, it becomes feasible to consider using fibers instead of threads to handle request processing.

Note: All of this performance work was done prior to the addition of filters to Apache 2.0. mod_file_cache, et. al. will need to be re-implemented to conform to the new filter API. The performance improvements should not be adversely impacted by the filter architecture.

Enhancements to mod_file_cache include support dynamic cache loading and automatically garbage collecting files that change on disk.

Where to find this write-up and presentation

This presentation can be found on-line at http://www.wstoddard.com/ac2keurope.