Apache 2.0

Ryan Bloom

Why Go Beyond 1.3?

Apache 1.3 is a great web server which serves pages for the vast majority of the web, but there are things it can't do. Firstly, it isn't particularly scalable on some platforms. AIX processes, for example, are very heavy-weight and a small AIX box serving 500 concurrent connections can become so heavily loaded that it can be impossible to telnet to it. In situations like this, using processes is not the right solution: we need a threaded web server.

Apache is renouned for being portable as it works on most POSIX platforms, all versions of Windows, and a couple of mainframes. However, like most good things, portability comes with a price which in this case is ease of maintenance. Apache is reaching the point where porting to additional platforms is becoming more difficult. In order to give Apache the flexibility it needs to survive in the future, this problem must be resolved by making Apache easy to port to new platforms. In addition, Apache will be able to use any specialised APIs, where they are available, to give better performance.

Multiple-Processing Modules (MPM)

The original reason for creating Apache 2.0 was scalability, and the first solution was a hybrid web server; one that has both processes and threads. This solution provides the reliability that comes with not having everything in one process, combined with the scalability that threads provide. The problem with this is that there is no perfect way to map requests to either a thread or a process.

On platforms such as like Linux, it is best to have multiple processes each with multiple threads serving the requests so that if a single thread dies, the rest of the server will continue to serve more requests. Other platforms such as Windows don't handle multiple processes well, so one process with multiple threads is required. Older platforms which do not have threads also had to be taken into account. For these platforms, it is necessary to continue with the 1.3 method of pre-forking processes to handle requests.

There are multiple ways to deal with the mapping issue, but the cleanest is to enhance the module features of Apache. Apache 2.0 sees the introduction of 'Multiple-Processing Modules' (MPMs) - modules which determine how requests are mapped to threads or processes. The majority of users will never write an MPM or even know they exist. Each server uses a single MPM, and the correct one for a given platform is determined at compile time.

What MPMs are available?

There are currently five options available for MPMs. Their names will likely change before 2.0 ships, but their behaviours are basically set. All of the MPMs, except possibly the OS/2 MPM, retain the parent/child relationships from Apache 1.3. This means that the parent process will monitor the children and make sure that an adequate number are running.

`PREFORK`

This MPM mimics the old 1.3 behaviour by forking the desired number of servers at startup and then mapping each request to a process. When all of the processes are busy serving pages, more processes will be forked. This MPM should be used for older platforms, platforms without threads, or as the initial MPM for a new platform.

`PMT_PTHREAD`

This MPM is based on the PREFORK MPM and begins by forking the desired number of child processes, each of which starts the specified number of threads. When a request comes in, a thread will accept the request and serve the response. If most of the threads in the entire server are busy serving requests, a new child process will be forked. This MPM should be used on platforms that have threads, but which have a memory leak in their implementation. This may also be the proper MPM for platforms with user-land threads, although there has not been enough testing at this point to prove this hypothesis.

`DEXTER`

This MPM is the next step in the evolution of the hybrid concept. The server starts by forking a static number of processes which will not change during the life of the server. Each process will then create the specified number of threads. When a request comes in a thread will accept and answer the request. At the point where a child process decides that too many of its threads are serving requests, more threads will be created. This MPM should be used on most modern platforms capable of supporting threads. It should create the lightest load on the CPU while serving the most requests possible.

PERCHILD

This MPM is based on the Dexter MPM. The biggest difference is that each child process can be assigned a different user and group. Each virtual host is then assigned to a set of child processes. If a child accepts a request for a virtual host that has been assigned to a different child process, then the request is passed to the correct child. This ensures that all CGI scripts are executed as the correct user. Apache 1.3 accomplished this using suexec, but suexec only worked for true CGI scripts, PHP and mod_perl scripts were not effected. With the PerChild MPM all requests that are served for a given virtual host use the correct user and group id.

`WINNT`

This MPM is designed for use on Windows NT. Before Apache 2.0 is released, it will also be made to work on Windows 95 and 98 although, just like Apache 1.3, it is unlikely to be as stable as on NT. This MPM creates one child process, which then creates a specified number of threads. When a request comes in it is mapped to a thread that will serve the request.

`OS/2`

This MPM is designed for use on OS/2. It is purely threaded, and removes the concept of a parent process altogether. When a request comes in, a thread will serve it properly, unless all of the threads are busy, in which case more threads will be created.

Multi-processing modules are designed to work behind the scenes and do not interfere with requests in any way. In fact, its only function is to map the request to a thread or process. One advantage of this technique is that each MPM can define its own directives. This means that if you are using a PREFORK MPM, you won't be asked how many threads you want per server, or if you are using the WINNT MPM, you won't need to specify the number of processes.

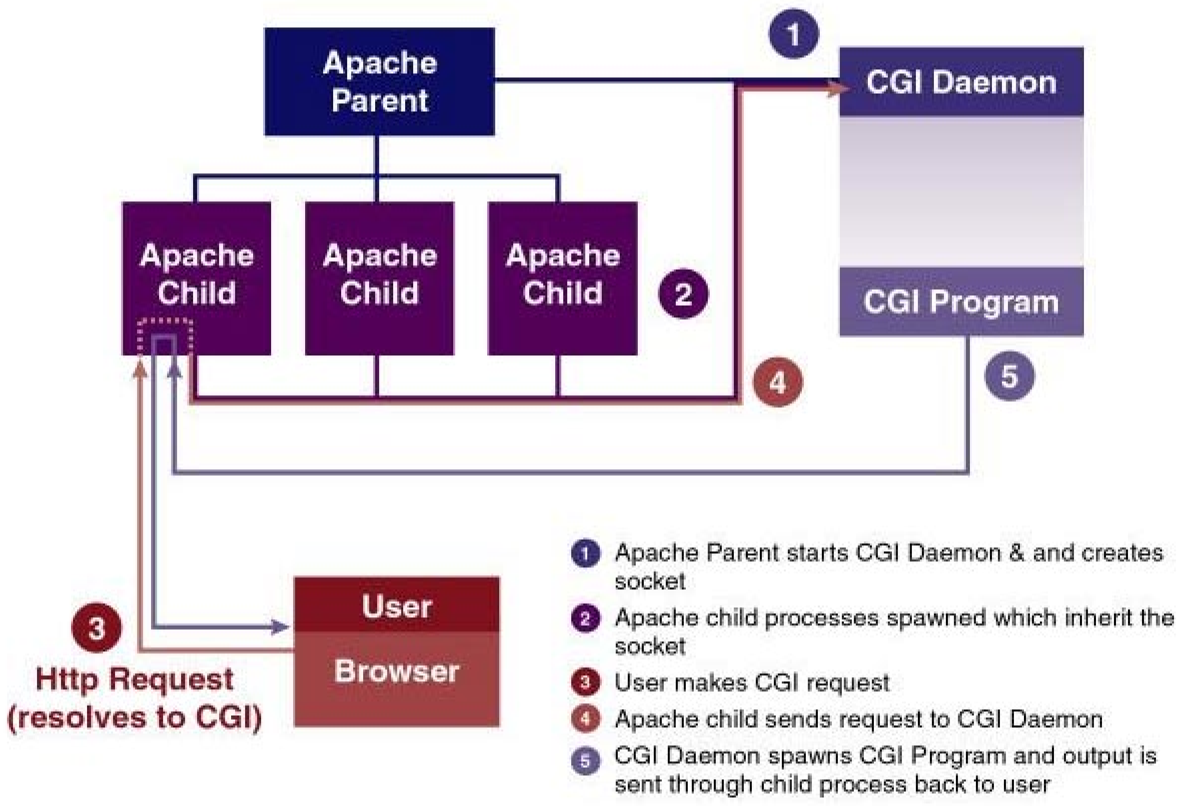
Protocol Modules

Apache 2.0 can also be thought of as a server framework as well as a web server. There is a new module type, called a protocol module, that allows Apache to server requests other than HTTP requests. Currently only HTTP and echo protocol servers have been written, but others should be possible. The point of protocol modules is to take advantage of the reliability, flexibility, and robustness of Apache while serving requests using multiple protocols. Each protocol module inherits from Apache the use of virtual hosts, and protocols can be enabled and disabled on a per-virtual host basis. At the end of this paper is an example protocol module, mod_echo. This is a very simple protocol module, that listens on a single port, and echos everything sent to that port back to the client.

New Method for serving CGI scripts.

When work began on 2.0, the developers discovered an interesting problem with CGI scripts and a hybrid thread/process Apache. Many Unix platforms do not fork threaded processes correctly. They will fork the entire process, copying all of the data for all of the threads, even though only one thread is active in the new process. This causes the performance of CGI scripts to be much worse than the performance of CGI scripts run by Apache 1.3. To solve this problem, the Apache developers created a new CGI module, mod_cgid.

This module creates a CGI daemon that is created when Apache is started. Each child process has a connection to this daemon. Whenever a child process receives a request for a CGI script, that child sends the request to the CGI daemon. The CGI daemon then creates the new CGI process, and passes the pipe back to the original child process. From there, CGI processing continues as normal. Because the CGI daemon is created in the Apache parent process, which is always a single-threaded process, creating the CGI processing doesn't suffer from the same problem as if the process was created by the child process.


1
Apache Parent
CGI Daemon
Apache Child
Apache Child
Apache Child
2
CGI Program
4
5
3
Http Request (resolves to CGI)
User
Browser
1 Apache Parent starts CGI Daemon & and creates socket
2 Apache child processes spawned which inherit the socket
3 User makes CGI request
4 Apache child sends request to CGI Daemon
5 CGI Daemon spawns CGI Program and output is sent through child process back to user

Will Apache 1.3 Modules work?

Modules written for 1.3 will not work with 2.0 without modification. There are many changes which will be documented by the time 2.0 is released. Many of these changes will be covered in the talk about writing Apache 2.0 modules so this talk will just cover those changes very briefly.

In Apache 1.3, each module uses a table of callback routines and data structures. Instead of using this table to specify which functions to use when processing a request, 2.0 modules will have a new function to register any callbacks needed.

In the past, new features have been added to subsequent releases of Apache which required the callback table to be expanded causing existing modules to break. In 2.0, each module is able to define how many callbacks it wants to use instead of using a statically defined table with a set number of callbacks. If the Apache Group decides to add callbacks in the future, the changes are less likely to affect existing modules.

Many things have been abstracted in Apache 2.0 and there are many new functions available. This means it will no longer be possible to access most of the internals of Apache data structures directly. For example, if a module needs access to the connection in order to send data to the client, it will have to use the provided functions rather than access the socket directly.

Apache 2.0 has also added I/O filtering. This gives Apache modules the ability to modify the output of any module that was run previously. One of the features most requested in Apache is the ability for mod_include to parse the output of mod_cgi. Filtered I/O allows for this. The design for filtered I/O had a few requirements. The biggest design constraint was that it had to be possible to write performance aware filters. The Apache developers do not want to force filter writers to consider performance, but at the same time, we wish to make it very easy to allow filter writers to take performance into consideration.

This was achieved with the use of bucket brigades. Bucket brigades are lists of small chunks of data. Each chunk of data has properties associated with it. For example, CGI scripts always produce bucket brigades of one element, a pipe. When the next filter wants to read from the pipe, it calls the pipe's read function, which splits the pipe bucket into two buckets, one with the data read from the pipe, and the other with the original pipe. This allows filters to take advantage of zero copy and sendfile when possible. For example, if a module adds a small text string that is allocated out of stack space to the end of a very large chunk of text, all of that data can be sent out with a single call to writev. In Apache 1.3, all of this data would have been copied to a single buffer and written using a call to write. This requires copying data that didn't need to be copied. Sendfile is another example of a useful performance tweak. Anytime Apache is going to serve data from a file, it is better to send that file over the network using sendfile. Using a file bucket, Apache 2.0 can determine if we are sending a file, and if so we can use sendfile.

New Configuration Tree

The configuration setup in Apache 2.0 has also changed drastically. Apache 2.0 uses a new configuration tree setup when reading the configuration file. Basically, as Apache reads the configuration file, it builds a tree structure representing the configuration for this server. Once the configuration has been read, modules have an opportunity to modify the configuration tree. Then, the tree is walked, and the configuration is activated for the tree. This has some useful side effects. It is much easier now for modules to modify the configuration because the internal format is well defined. Also, mod_include becomes much simpler, because it just walks the tree and writes the directives found back to the client.

However, this also has some interesting side effects. The Apache configuration is very dependent on the order in which directives appear. For example, if we wish to load a dynamic module, that must be finished before we can handle any of the directives implemented by the module. To handle this, Apache 2.0 has added a special flag to the req_overrides field of the directive definition, EXEC_ON_READ. Any directive defined as EXEC_ON_READ is executed as it is read from the configuration file. This should only be used by those directives that control how Apache interprets the configuration file.

New Hook Mechanism

One of the biggest drawbacks to Apache 1.3, is that modules were always treated as single elements. This meant that if there were two modules in a given Apache installation, mod_mime and mod_mime_magic, whichever was inserted into the server first would run all of its hooks before the other. For example, assume mod_mime was installed in the server first, all of its hooks would run before the corresponding hook for mod_mime_magic. This can lead to interesting configurations. In Apache 2.0 module's can be sorted on a per-hook basis. This means that if mod_mime's type_checker needs to run before mod_mime_magic's type checker, but the initialization phase has the reverse requirements, Apache 2.0 can handle those requirements.

The order of modules is done by the module when it registers the hook. Basically, each module specifies which hooks it wishes to register a function for. At that time, the module can specify if there are any other order dependancies that it knows about with other modules for that hook.

The Apache Portable Run-Time (APR)

APR was originally designed as a way to combine code across platforms. There are some sections of code that should be different for different platforms as well as sections of code that can safely be made common across all platforms.

Apache on Windows currently uses POSIX functions and types that are non-native and non-optimised for communicating across a network. By replacing these functions and types with the Windows native equivalent there has been a significant performance improvement. For example, spawning CGI processes is very confusing in Apache 1.3 because Unix, Windows, and OS/2 all handle spawning in different ways. By using APR, the logic can be combined for spawning CGI processes, decreasing the number of platform-specific bugs that are introduced later.

APR will make porting Apache to additional platforms easier. With a fully implemented APR layer any platform will be able to run Apache. APR is small and well defined and once it is fully integrated into Apache, will change very little in the future. Apache has never been well defined for porting purposes as there was too much code to make porting a simple task. In addition, the code was originally designed for use on Unix, which made porting to non-POSIX platforms very difficult. With APR, all a developer needs to do is implement the APR layer. APR was designed with Windows, Unix, OS/2, and BeOS in mind and is more flexible as a result.

APR acts as the abstraction layer in Apache 2.0. To allow the use of native types for the best performance, APR has unified functions such as sockets into a single type which Apache will then use independently of the platform. The underlying type is invisible to the Apache developer, who is free to write code without worrying about how it will work on multiple platforms.

MOD_ECHO

```
/* ====================================================================
 * The Apache Software License, Version 1.1
 *
 * Copyright (c) 2000 The Apache Software Foundation.  All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. The end-user documentation included with the redistribution,
 *    if any, must include the following acknowledgment:
 *       "This product includes software developed by the
 *        Apache Software Foundation (http://www.apache.org/)."
 *    Alternately, this acknowledgment may appear in the software itself,
 *    if and wherever such third-party acknowledgments normally appear.
 *
 * 4. The names "Apache" and "Apache Software Foundation" must
 *    not be used to endorse or promote products derived from this
 *    software without prior written permission. For written
 *    permission, please contact apache@apache.org.
 *
 * 5. Products derived from this software may not be called "Apache",
 *    nor may "Apache" appear in their name, without prior written
 *    permission of the Apache Software Foundation.
 *
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 * ====================================================================
 *
 * This software consists of voluntary contributions made by many
 * individuals on behalf of the Apache Software Foundation.  For more
 * information on the Apache Software Foundation, please see
 * <http://www.apache.org/>.
 *
 * Portions of this software are based upon public domain software
 * originally written at the National Center for Supercomputing Applications,
 * University of Illinois, Urbana-Champaign.
 */

#include "ap_config.h"
#include "ap_mmn.h"
#include "httpd.h"
#include "http_config.h"
#include "http_connection.h"
#include "mod_echo.h"

AP_HOOK_STRUCT(
    AP_HOOK_LINK(post_echo_read)
)
AP_IMPLEMENT_HOOK_VOID(post_echo_read, (char **c, int *l), (c,l));

API_VAR_EXPORT module echo_module;

typedef struct
    {
    int bEnabled;
    } EchoConfig;

static void *create_echo_server_config(apr_pool_t *p,server_rec *s)
    {
    EchoConfig *pConfig=apr_pcalloc(p,sizeof *pConfig);

    pConfig->bEnabled=0;

    return pConfig;
    }

static const char *echo_on(cmd_parms *cmd, void *dummy, int arg)
    {
    EchoConfig *pConfig=ap_get_module_config(cmd->server->module_config,
                                             &echo_module);
    pConfig->bEnabled=arg;

    return NULL;
    }

static int process_echo_connection(conn_rec *c)
{
    char *buf = apr_palloc(c->pool, 1024);
    EchoConfig *pConfig=ap_get_module_config(c->base_server->module_config,
                                             &echo_module);

    if(!pConfig->bEnabled)
```

```
            return DECLINED;

    for( ; ; ) {
            apr_ssize_t r, w;
        (void) ap_bread(c->client,buf,1024,&r);
            if (r <= 0)
                break;
        ap_run_post_echo_read(&buf, &r);
            (void) ap_bwrite(c->client,buf,r, &w);
            if (w != r)
                break;
            ap_bflush(c->client);
    }
    return OK;
}

static const command_rec echo_cmds[] =
{
    AP_INIT_FLAG("ProtocolEcho", echo_on, NULL, RSRC_CONF,
              "Run an echo server on this host"),
    { NULL }
};

static void register_hooks(void)
{
    ap_hook_process_connection(process_echo_connection,NULL,NULL,AP_HOOK_MIDDLE);
}
API_VAR_EXPORT module echo_module = {
    STANDARD20_MODULE_STUFF,
    NULL,                        /* create per-directory config structure */
    NULL,                        /* merge per-directory config structures */
    create_echo_server_config,   /* create per-server config structure */
    NULL,                        /* merge per-server config structures */
    echo_cmds,                   /* command apr_table_t */
    NULL,                        /* handlers */
    register_hooks          /* register hooks */
};
```