Scaling Apache 2.x beyond 20,000 concurrent downloads

Colm MacCárthaigh
colm@stdlib.net

21st July 2005

Abstract

In 2002, Ireland's National Research and Education Network, HEAnet, decided to overhaul its mirroring service - ftp.heanet.ie. The re-launched service, using Apache 2.0, quickly attracted attention as it became the first official Sourceforge mirror not run by the OSDN group.

Due to a combination of Sourceforge being entirely HTTP-based, and ftp.heanet.ie being responsible for approximately 80% of all Sourceforge downloads between April 2003 and April 2004, sustaining over 20,000 concurrent connections (current record is 27,317) from a single Apache instance has become a regular occurrence.

This paper, and accompanying talk, share the lessons learned and the techniques used to achieve this scalability.

Contents

1 Introduction

> "Mirror mirror, on the wall, who is the fairest of them all?"
>
> The Wicked Stepmother

Generally speaking, supporting tens of thousands of simultaneous requests from a single webserver is not what you're supposed to do. Usually, such high demands on a service mean there should be some form of sensible load-balancing; a webserver cluster. There are times though when budget or other project constraints may demand exceptional levels of performance from single machines, or then again it can be worth doing simply because it's cool.

When implementing our Mirror service, it was much more economical - for reasons I'll explain in section 5, for us to run ftp.heanet.ie from a single machine and to expend effort tuning the system for greater scalability. This approach almost certainly isn't for everyone, but hopefully the tuning and scalability lessons learned will prove of benefit to others - to reduce server load and increase responsiveness.

1.1 What ftp.heanet.ie is

ftp.heanet.ie is HEAnet's National Mirror Server for Ireland. Currently mirroring over 50,000 projects, it is a popular source of content on the internet. It serves content via HTTP, FTP and RSYNC all available via IPv4 and IPv6.

Re-launched in 2002 on new hardware, it has attracted a large number of users. During major software releases it has frequently exceeded 20,000 simultaneous HTTP downloads and has saturated a gigabit of connectivity in production. We believe it to be one of the busiest single webservers in the world.

1.2 What ftp.heanet.ie is not

ftp.heanet.ie does not serve much dynamic content. Apart from directory indexes, and the content available in the root and about URIs, there is no dynamic content. Although some mirror servers host dynamic content for projects which demand it (The PHP and eclipse projects for example require php of mirrors), we choose not to mirror these projects.

This is primarily for security reasons. We do not believe the risk of hosting such code on a server with multiple gigabit/sec of internet connectivity and terabytes of storage is worth it.

Most webservers increasingly do serve a large amount of dynamic content however. This paper should still prove useful to administrators of such servers, as the topics covered are easily applicable to those systems.

It would not be safe however, to assume that the scalability we have achieved on a single system serving only static content is portable to a system serving primarily dynamic content. On such

systems it generally the applications providing the dynamic content which are the source of bottlenecks and those that need to be tuned.

1.3 Motivation

HEAnet is Ireland's National Research and Education Network, we serve as both an ISP and a research platform for Ireland's Universities and Institutes of Technology. Historically speaking, Mirroring services existed to save ISP's bandwidth; by archiving a large amount of popular content locally, users would download from there and save expensive and scarce international transit bandwidth.

This, however is not the motivation behind the re-launched ftp.heanet.ie. The new service actually increased our international transit bandwidth usage due to its popularity. This is not a cause for concern; beyond DSL and dial-up links, IP connectivity is universally symmetrical - a 1 Gigabit/sec link can ship 1 Gigabit/sec in each direction simultaneously. As HEAnet was a net importer of traffic and export bandwidth rarely exceed 10% of link-capacity, adding more would not contend with customer traffic. (We also implemented Committed Access Rate-limiting to ensure this).

These days, HEAnet has at least 10 Gigabit/sec of international bandwidth to facilitate academic research and provide a resilient service. As such, it would be impossible for ftp.heanet.ie to become a source of contention on our network.

The primary motivation behind ftp.heanet.ie was to make customers' downloads faster. As HEAnet deployed gigabit links to customers, the main constraint for download speed was TCP bottlenecks caused by the round-trip times to off-network sites. Customers really appreciate an ISO download taking 40 seconds instead of 30 minutes.

Beyond this though, there are other benefits. HEAnet offers a range of services, and operating and developing a challenging service such a large-scale mirror gave us a well-used platform on which to experiment and develop configurations that would later be applied elsewhere.

A secondary but nonetheless important motivation was to give something back to the Open-Source community. The academic networking community has a strong tradition of using Open-Source software and HEAnet is no exception. Although HEAnet contributes to various projects in the form of employee time, patches, development effort and so on, ftp.heanet.ie was seen as another way to help out.

Another benefit is that having terabytes of traffic per day across the network serves as a great way to test the network, ensure its stability and find bugs quickly. As such, ftp.heanet.ie is as much a part of network development as it is a systems operation.

1.4 The Numbers

At the time of writing (May 2005), ftp.heanet.ie;

- can sustain at least 27,000 simultaneous HTTP downloads

can saturate a gigabit interface at the same time

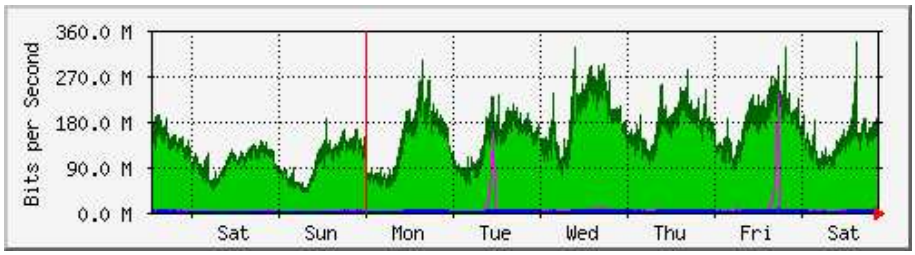
has roughly 3.2 million downloads per day

ships about 3.5 Terabytes of content per day

mirrors over 50,000 projects

has 3.7 Terabytes of unique content

has circa 6 million unique files



Currently, approximately 90% of downloads are via HTTP, the remainder being via FTP and a negligible amount over rsync. For a time (purely for nostalgia), ftp.heanet.ie ran a gopher service, it had a total of 12 downloads via the gopher protocol.

ftp.heanet.ie does not offer carrier or even enterprise-grade uptime. Although unscheduled down-time is much less frequent than previously, there is a 5 minute outage (for a reboot) every 7 weeks on average. Since upgrading to version 2.6.11 of the Linux kernel no such outage has been observed, though it remains to see how long this will last. The previous outages seem to be attributable to the Linux 2.6 kernel's over-zealous process-killing (even killing init once).

2 Benchmarking

> "No one tests the depth of a river with both feet."
>
> African Proverb

Before going on to the design and tuning of a system in detail, it is necessary to introduce the benchmarking techniques used. Rather than apply random configurations and just hope, it is usually better to apply some of the scientific method to systems design (or rather engineering). Benchmarking is an integral component of this.

Apache has many aspects which can be usefully benchmarked; as a webserver its primary purpose is to take content you have locally and to serve it to the network using TCP. Local content can be either static content stored on the filesystem, or dynamic content generated by scripts or modules. Either way, how quickly files can be read from the filesystem is of critical importance.

For the purposes of a mirror server, this is almost the only type of content served. Thus it is essential that the filesystem and filesystem-options chosen be fast and efficient.

2.1 Webserver benchmarking

Since this paper is about scaling a webserver, the most important benchmark is that of the webserver itself. Ultimately you can (and indeed should) use this benchmark to measure the impact of any changes to the system, after all it should always filter down into faster webserving!

There are a great many tools for benchmarking webservers, ranging from a simple wget/curl to a full Specweb [spe99] evaluation. For the most part though, Apachebench - which comes with Apache - and httperf [MJ] (and the autobench wrapper script) were sufficient for our needs.

For our benchmarks, we usually invoke Apachebench with a concurrency level of 100 and at least 1000 requests, though usually more. To ensure the consistency of the tests, we keep several files of known-sizes (containing random data) on the server for use during testing. Currently the files are 100, 1000, 2000 and 10000 bytes in size.

```
colmmacc@byron:~$ ab -q -c 100 -n 1000 http://ftp.heanet.ie/pub/heanet/100.txt
This is ApacheBench, Version 1.3d <$Revision: 1.65 $> apache-1.3
Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright (c) 1998-2002 The Apache Software Foundation, http://www.apache.org/

Benchmarking ftp.heanet.ie (be patient).....done
Server Software:        Apache/2.1.5-dev
Server Hostname:        ftp.heanet.ie
Server Port:            80

Document Path:          /pub/heanet/100.txt
Document Length:        100 bytes

Concurrency Level:      100
Time taken for tests:   5.989 seconds
Complete requests:      1000
```

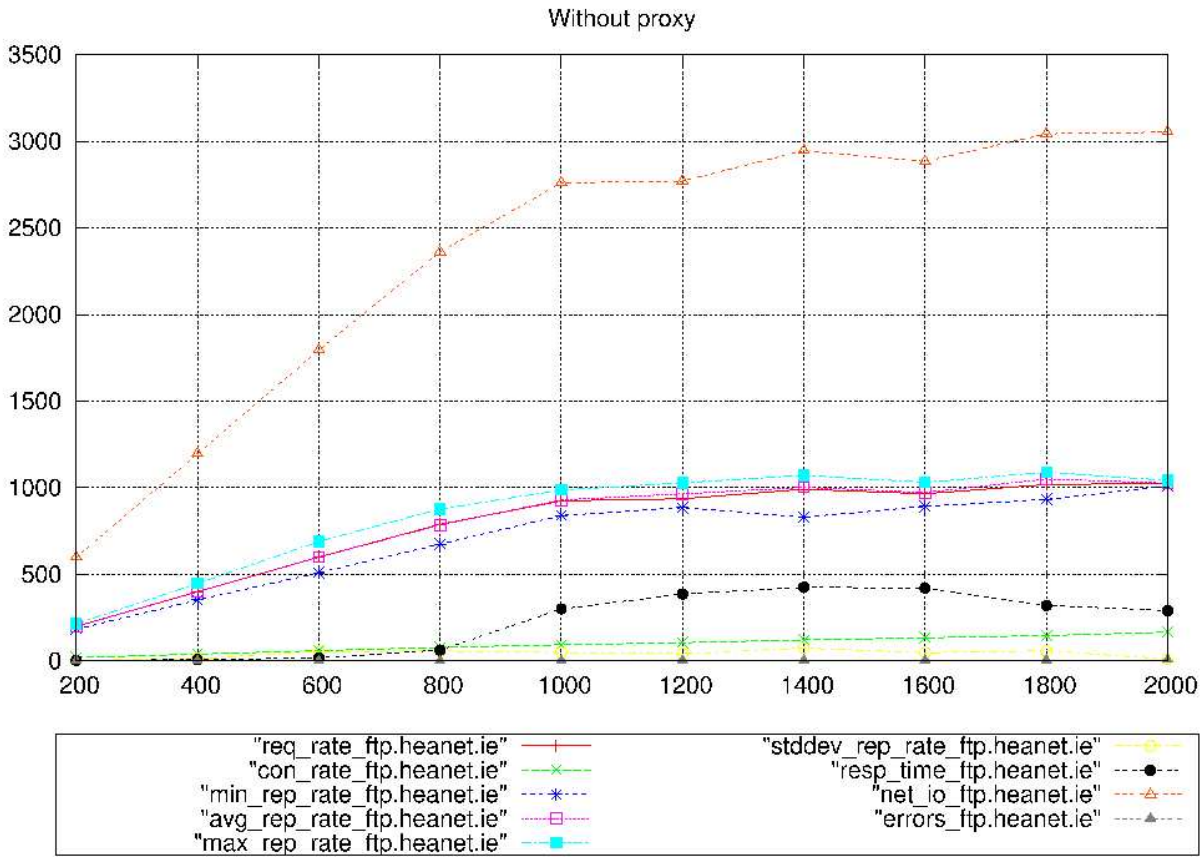
```
Failed requests:        0
Broken pipe errors:     0
Total transferred:      372000 bytes
HTML transferred:       100000 bytes
Requests per second:    166.97 [#/sec] (mean)
Time per request:       598.90 [ms] (mean)
Time per request:       5.99 [ms] (mean, across all concurrent requests)
Transfer rate:          62.11 [Kbytes/sec] received

Connnection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0     0    0.2      0     3
Processing:     4   544  561.8    288  1793
Waiting:        2   544  561.9    288  1793
Total:          4   545  561.6    288  1793
WARING: The median and mean for the initial connection time are not within a normal deviation
        These results are propably not that reliable.

Percentage of the requests served within a certain time (ms)
  50%    288
  66%    772
  75%   1018
  80%   1081
  90%   1466
  95%   1561
  98%   1589
  99%   1786
 100%   1793 (last request)
```

The above invocation of Apachebench represents the performance of ftp.heanet.ie , while it was also serving approximately 5,000 requests.

When using the autobench script for httperf, much more granular results are given. Autobench iterates through incrementing passes of httperf to try and loadtest the webserver. Invoked as follows;

```
autobench --single_host --host1 ftp.heanet.ie --uri1 / --quiet     \
          --low_rate 20 --high_rate 200 --rate_step 20 --num_call 10 \
          --num_conn 5000 --timeout 5 --file with.tsv
```

autobench runs several tests ranging from 20 requests per second to 200 per second, with 5,000 connections per iteration (it takes some time to fully run). Autobench supplies a “bench2graph” utility which generates useful graphs such as the below.



Graphs from autobench are particularly useful for identifying any errors the webserver may be producing, but also serve as a good graphical means for judging and illustrating performance.

2.2 Filesystem benchmarking

There are various utilities designed for filesystem benchmarking on Unix. Iozone, postmark and bonnie++ are three of the most common. Postmark (from Network Appliances) though excellent, is based around small filesizes typical of Mail and News servers. As such iozone and bonnie++ tended to be the tools most useful for our benchmarking.

bonnie++'s benchmarking, though less extensive than iozone's, gives more than enough detail to compare filesystems with each other for suitability. Iozone did prove more useful for benchmarking the more minute tuning of filesystems, once selected however.

bonnie++ is relatively simple to use, and needs only for a directory on the target filesystem to be writable by the user you wish to run bonnie++ as (it will refuse to run as root). When run, it creates a series of files (adding up to twice the amount of system memory in size) and generates output of the form;

```
Version 1.02b       ------Sequential Output------ --Sequential Input- --Random-
                    -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine        Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP  /sec %CP
cassandra       24G 22479  97 53690  36 29763  14 23733  96 62687  13 147.7   0
```

```
      ------Sequential Create------ --------Random Create--------
      -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
files  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP
   16 32270  97 +++++ +++ +++++ +++ 31053  99 +++++ +++ 32164  92
```

For the purposes of a webserver the most important number to know is how fast files can be read. That's the "Block" section under "Sequential Output". The "%CP" also gives an idea of how CPU-intensive the read operations were during benchmarking. In the above example, the filesystem read at 53690 Kbytes per second, which is the same as 430Mbit/sec.

Clearly the filesystem tested here would have no problem saturating a 100Mbit link on its own, but would need some help saturating a gigabit interface.

2.3 VM and Scheduler benchmarking

Unfortunately, there are really no generic tools for benchmarking virtual memory managers and schedulers. Typically they are benchmarked by measuring the performance of a resource intensive task, such as a highly-loaded webserver, compiling a kernel, encoding a movie and so on.

While to a large extent we took the approach of measuring Apache's performance directly following scheduler and VM changes, it is also useful to model the system in a more basic and fundamental way so that the impact of changes can be more easily determined and illustrated.

Depending on the MPM used, Apache's load can be modelled very differently. However our main aim was to improve the rate at which a single Apache thread/process could read and serve files. As we'll see later, zero-copy, where the system tries to read in files at the same rate at which it writes them - without using memory to buffer differences - is very desirable when serving content.

In an effort to synchronise the rates at which the kernel and Apache read files (by changing "fs/read_write.c" from the Linux kernel and "server/core.c" from Apache, we wanted to determine the most efficient read size for our filesystem/disks.

To that end, we produced a script - which using the debian version of dd [1] - would scan through a range of read buffer sizes and allow us to graph the result;

```
#!/bin/sh

STARTNUM="1"
ENDNUM="102400"

# create a 100 MB file
dd bs=1024 count=102400 if=/dev/zero of=local.tmp

# Create the record
rm -f record
touch record
```

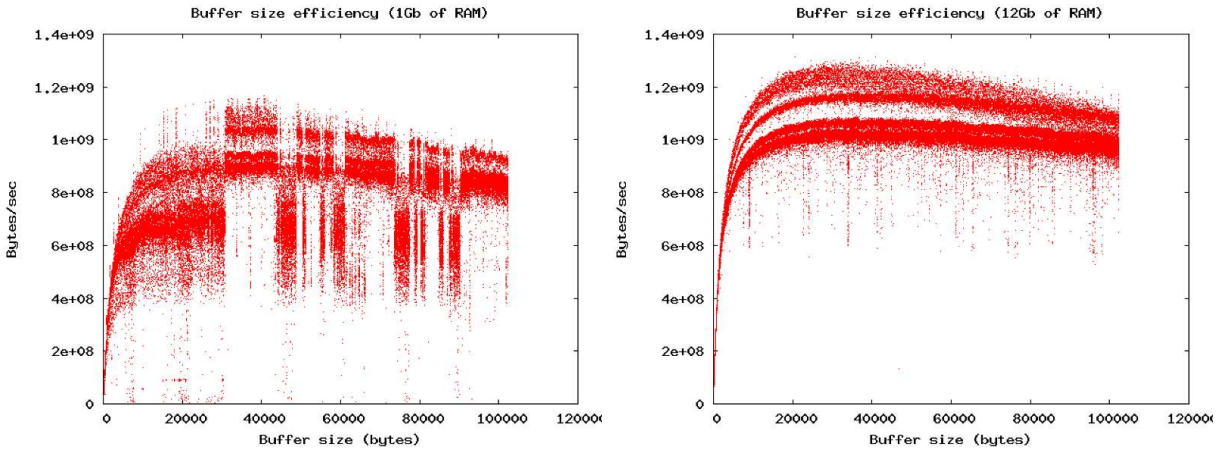
[1] Debian's version of dd includes dd-performance-counter.patch which outputs bytes per second statistics

```
# Find the most efficient size
for size in 'seq $STARTNUM $ENDNUM'; do
                ./dd bs=$size if=local.tmp of=/dev/null        2>> record
done

# get rid of junk
grep "transf" record | awk '{ print $7 }' | cut -b 2- | cat -n | \
while read number result ; do
                echo -n $(( $number + $STARTNUM - 1 ))
                echo " " $result
done > record.sane
```

As the graphs below show, it turns out the most optimum read buffer sise is about 20Kb. We found this fairly surprising as most unix utilities use much smaller buffer sizes, usually 1024 or 4096 bytes. Unfortunately however 20Kb is far too large for true zero-copy when dealing with networked applications, due to the typical 1500 byte MTU size.

Although the script was designed to measure the optimum read buffer size it proved to be even more useful as a measure of both VM and scheduler performance. The Linux virtual memory manager caches files aggressively, indeed it is not even possible to tell it not to cache files. Substantial performance benefits can be had merely by reading a file twice. As such, the continuous dd's are actually measuring the VM performance rather than raw read speed from the filesystem - thus the height of the curve is a measure of VM performance.

By running the tests on a non-idle system with some load (Apache serving clients, a kernel compiling, and so on), the scheduler is forced to regulate the ammount of time-slices (jiffies) available to the dd instances. Across a large number of dd's this leads to "jitter" in the graph. Thus graph-jitter, or the second-derivative of the graph function, is an effective measure of the scheduler performance. The smoother and higher the graph, the better the aggregate performance of both the scheduler and the memory manager.



The first graph above is a graph of the script behaviour on canyonero.heanet.ie, with 1Gb of memory and the second graph is cassandra.heanet.ie with 12Gb of memory. There is a very clear improvement in the second graph.

Notice also the bandation of the graphs; such bandation is typical of processes which involve harmonic and non-harmonic modes. Certain buffer sizes that are nearly, but not quite, efficient

will be particularly inefficient and will have their own harmonics throughout the range.

3 Tuning Apache

> "Harpists spend 90 percent of their lives tuning their harps and 10 percent playing out of tune."
>
> Igor Stravinsky

3.1 Choosing an MPM

Apache 2.x has numerous available multi-processing modules, which offer differing ways to manage processes internal to Apache. On Unix at least, the two most worth considering are the prefork MPM and the worker MPM.

Without putting much thought into actually choosing an MPM, ftp.heanet.ie used the default, prefork MPM for quite some time. However after many repeated suggestions that the thread-based worker MPM may offer us benefits, we decided to evaluate it.

Unfortunately for us, the worker MPM would exit with a segmentation fault after at most 5 minutes of operation. The core dump seemed to be less than helpful for performing a useful backtrace and left us (and httpd-dev) none the wiser about a possible cause.

With the work of Aaron Bannert and Jeff Trawick, and using the new modules mod_forensic_id and mod_backtrace, this was eventually tracked down to a stack-overlay bug in the worker MPM that presumably few other servers had been busy enough to notice. A patch for this bug was incorporated into Apache 2.0.49 [2].

After incorporating the patch, we then ran our own benchmarks on the worker MPM, however for our load it reduced the number of requests per second that could be sustained. It did decrease the amount of memory used, so it does have its benefits (it can also significantly increase performance on platforms such as Solaris, which has lightweight process support). However as our bottom line was requests-per-second and memory is relatively inexpensive (compared to development time), we chose to continue using the prefork MPM.

We are continually re-evaluating this and occasionally re-measure the performance of the worker (and other) MPMs.

While evaluating the worker MPM we did have to examine how best to tune it. The worker MPM functions as a series of ordinary child processes which in turn are host to threads. However it is less than clear whether it is more efficient to have more children with a smaller number of threads per child or to have the converse.

After a large amount of experimentation and benchmarking, for our load we found that it was optimal to have 32 threads per child. Beyond this it seemed that management of the threads became a further overhead over that of managing ordinary processes.

Our final worker configuration;

[2] despite months of effort, this patch turned out to be one-line long, another lesson in how lines of code is an entirely useless metric of productivity

```
<IfModule worker.c>
        ServerLimit         900
        StartServers        10
        MaxClients          28800
        MinSpareThreads     25
        MaxSpareThreads     75
        ThreadsPerChild     32
        MaxRequestsPerChild 100
</IfModule>
```

the prefork MPM on the other hand is simpler to tune. The main consideration here is to ensure that the number of spare servers is such that there are enough child processes available to handle new requests when the rate of new connections exceeds the rate at which Apache can manage to create new processes.

For our load, we need only 10 spare servers, and as the autobench graphs show, we get no refused connections.

Our prefork configuration:

```
<IfModule prefork.c>
        StartServers        100
        MinSpareServers     10
        MaxSpareServers     10
        ServerLimit         50000
        MaxClients          50000
        MaxRequestsPerChild 2000
</IfModule>
```

We set a maximum number of requests per child mainly to protect against any possible memory leaks. On a system with several-tens-of-thousands of httpd processes even very small leaks will cause problems.

3.2 Static Vs DSO

Apache modules can be compiled in directly to one binary, or as dynamically-loaded shared-objects which are then loaded by a smaller binary. For our load, a small performance gain (measurable as about 0.2%) was found by compiling the modules in directly.

We did not go the extra step of linking the httpd binary entirely statically as any benefits would not justify the extra overhead of having to remember to recompile apache if any of the linked-libraries were upgraded for security reasons.

3.3 Configuration changes

As the highperformance.conf sample config which ships with Apache points out, turning off the use .htaccess files can greatly speed things up. The relevant part of our config being:

```
<Directory "/ftp/">
        Options Indexes FollowSymLinks
        AllowOverride None
        Order allow,deny
        Allow from all
        IndexOptions NameWidth=* +FancyIndexing \
                     +SuppressHTMLPreamble +XHTML
</Directory>
```

This however is not without its problems. When mirroring over 50,000 projects, some (including Apache itself) do in fact contain .htaccess files and require it.

As such, we regularly run "find ./ -type f -name .htaccess" to determine what projects need the access re-enabled. Initially, we produced a script that combined the contents of all these .htaccess files into one large configuration file;

```
#!/bin/sh

for file in `find /ftp/ -type f -name .htaccess`; do
    echo "<Directory `basename $file`>"
    sed 's/^/   /' $file
    echo "</Directory>"
    echo
done > combined-htaccess.conf
```

The individual .htaccess files changed infrequently enough that this did not cause problems for maintainers. We also manually reviewed the config file to confirm that no malicious directives were being introduced.

What did however prove a problem is that a large amount of projects symlink directories multiple times. As such, there were occasional problems when a user was accessing content through a newly-created symlink and the relevant directives would not be applied.

The overhead generated by constantly tracking these changes proved too much, and instead .htaccess was enabled on per-project basis, for example:

```
<Directory "/ftp/mirrors/www.apache.org/">
        IndexOptions FancyIndexing NameWidth=* FoldersFirst \
```

```
                    ScanHTMLTitles DescriptionWidth=*
        HeaderName HEADER.html
        ReadmeName README.html

        AllowOverride FileInfo Indexes
        Options Indexes SymLinksIfOwnerMatch
</Directory>
```

3.4 Sendfile

Sendfile is a system call available on many Operating Systems that enables programs to hand off the job of sending files out of network sockets to the kernel. Thus rather than wasting memory with a read buffer and having to worry about optimal buffer sizes, the kernel can optimise all of the reading, ideally in a so-called zero-copy manner.

Sendfile is enabled by default at compile-time if Apache detects that the system supports the call. The directive;

```
EnableSendfile On
```

enables sendfile within the Apache configuration.

If at all possible, use of the sendfile call will almost certainly boost performance and increase efficiency. Unfortunately in our case, Linux's sendfile implementation does not operate on our hardware (or any that we can find) without corrupting IPv6 sessions[3]. As HEAnet is committed to offering all production services over IPv6 as well as IPv4, simply turning off IPv6 was not an option for us.

One solution was to patch the code simply not to use sendfile if a request is being handled over IPv6. Indeed there are now patches included in the Apache Portable Runtime and proposed for httpd that would implement this more generally.

However as we had established that we could saturate our connectivity and handle very large numbers of users, we elected to live with the increased load.

3.5 Mmap

Following sendfile, the next best thing is Mmap (memory map). The use of Mmap can be enabled in an Apache config with;

```
EnableMmap On
```

[3]The bug is actually caused by TCP checksum miscalculations by almost all network interfaces, and Linux uses checksum offloading during use of the sendfile call. This was tracked down with the help of Joe Orton

Mmap support allows Apache to treat a file as if it were a contiguous region of memory, greatly speeding up the I/O by dispensing with unnecessary read operations.

Enabling Mmap support makes a great deal of difference to our performance, and allows us to serve files roughly 3 times quicker than we otherwise would be able to. Of course, if available, using sendfile would likely generate an even greater performance benefit. Using Mmap has no impact on IPv6 however.

3.6 mod_cache

mod_cache, with mod_disk_cache is one of most significant ways in which we have been able to boost our performance. Although we have roughly 3 terabytes and 6 million files of content, very little of it is popular at any given time. In a given 6-hour window only about 50,000 different files representing 10Gb of unique content is downloaded.

As our large file-systems are ultimately served by 7200 RPM IDE disks, it is generally a very good thing if we can reduce the load to these systems. To that end, we created a 36Gb RAID-0 container using two 18Gb 15k RPM SCSI disks. The combination of RAID-0 and 15k RPM disks makes the file-system on this container substantially quicker than the larger file-systems.

mod_disk_cache caches files in a configured area as they are being served for the first time. Repeated requests are served from this cache, avoiding the slower file-systems. mod_disk_cache comes with a fine sample-config, however we have changed ours to increase the CacheDirLevel[4] to 5 due to the large amount of files we were placing in the cache.

```
<IfModule mod_cache.c>
        <IfModule mod_disk_cache.c>
         CacheRoot /usr/local/web/cache/
         CacheEnable disk /
         CacheDirLevels 5
         CacheDirLength 3
        </IfModule>
</IfModule>
```

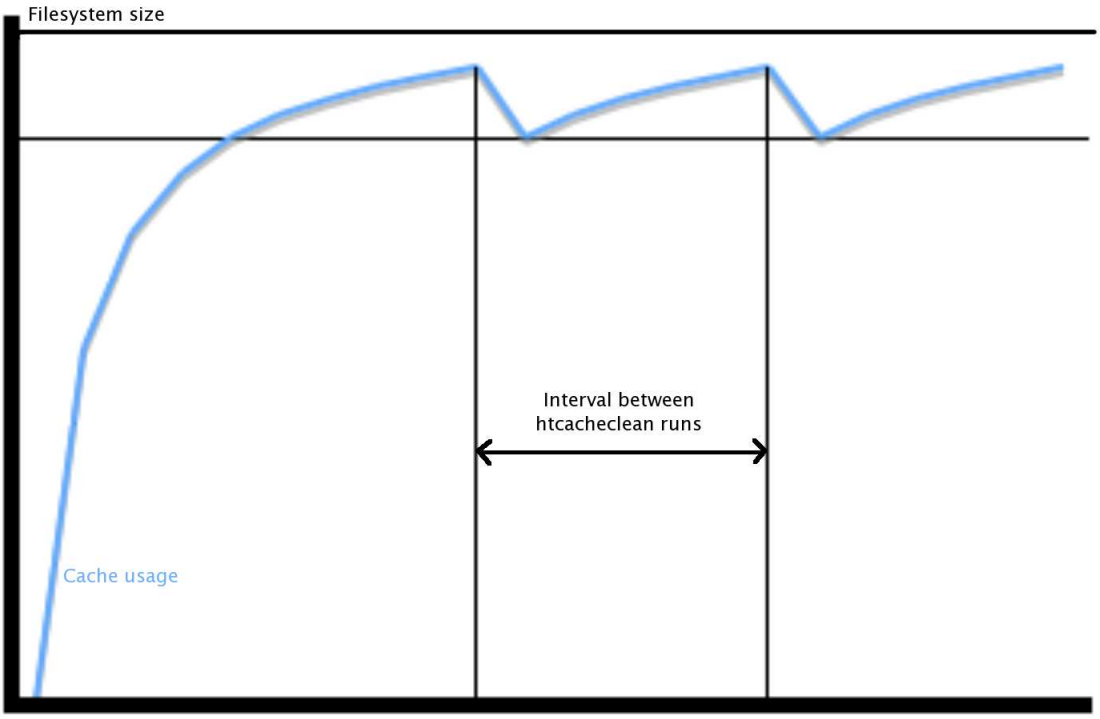
It should be noted that mod_disk_cache is still marked as experimental in Apache 2.0 and in a state of flux in Apache 2.1 (at the time of writing, mod_disk_cache appears to incorrectly handle 304 status responses) and is probably not yet suited for general use. However if time is taken to carefully monitor its behaviour it can be a source of great performance increases.

Another thing to note is that cleaning of the cache is not functional in Apache 2.0. For some time, we resorted to using a rather brutal combination of find, xargs and rm, however Apache

[4] mod_dir_cache stores cached files in a schema which uses directory and file names as a hash-table

2.1 now includes a working htcacheclean utility for maintaining the cache[5].

htcacheclean, run either from cron or as a daemon, will clean the cache only periodically and does not monitor it in real time. As such, it is important to ensure that the amount by which the cache increases in size during the interval between htcacheclean runs is not greater than the spare space on the file-system after htcacheclean has pruned the cache to its target level.



3.7 Compile options

For reference, our configure options are:

```
CFLAGS="-D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE"; export CFLAGS
"./configure" \
"--with-mpm=prefork" \
"--prefix=/usr/local/web" \
"--enable-cgid" \
"--enable-rewrite" \
"--enable-expires" \
"--enable-cache" \
```

[5]The author is considering writing an implementation of mod_disk_cache that would use innd's CNFS algorithm for using a disk as a round-robin database directly, but has not yet found the time

```
"--enable-disk-cache" \
"--without-sendfile"
```

mod_rewrite is used by a number of projects hosted, as is mod_expires. In general our approach has been to compile as little into the httpd binary as necessary, to reduce the amount of memory used on the system. The CFLAGS exported enable us to serve files over 2Gb in size, of which we host a number.

4 Tuning the Operating System

> "Feed the musician, and he's out of tune"
>
> George Crabbe

A webserver does not run in isolation, but is supported by an Operating System which has huge implications for the performance. Apache running on Solaris is a radically different beast to Apache running on FreeBSD to Apache running on Linux and so on.

With ftp.heanet.ie, we chose to remain with Debian GNU/Linux 3.0, our standard platform. As such this section concerns the effort we did to tune Linux to our needs. Similar work could be repeated on other systems and they may very likely turn out to be better (we suspect FreeBSD may be so), but we have not yet done this work.

4.1 Choosing and tuning filesystems

Choosing an efficient and fast filesystem is of critical importance to the operation of a webserver. A large amount of Apache's runtime is spent reading files from the filesystem, and it is very definitely in our interests for this to be as fast as possible.

At the time ftp.heanet.ie was commissioned we had 3 main choices of filesystem on Linux; ext2, ext3 and XFS. At that stage XFS was distributed in patch form by SGI. We discounted ext2 immediately as we were not willing to use a non-journaling filesystem. The thought of waiting on a lengthy fsck of a 2 terabyte filesystem is far too uncomfortable to dwell on.

We experimented with several ext3 and XFS filesystems, each made with various differing options, XFS consistently came out with higher numbers by a margin of 20%. Additionally, mkfs.xfs turned out to be the only version of mkfs that would even build a 2 terabyte filesystem at the time. Though this was later corrected. mkfs.xfs does not have many options and the options that it does have auto-detect sane defaults. It was thus invoked;

```
mkfs.xfs /dev/sdc1
```

We did however supply some mount options for the filesystem;

```
dev/sdc1        /ftp            xfs    rw,noatime,logbufs=8   0 1
```

"noatime" is perhaps the single easiest way to dramatically increase filesystem performance for read operations. Ordinarily when a file is read, Unix-like systems update the inode for the file with this access time - so that the time of last access is known. This operation however means that read operations also involve writing to the filesystem - a severe performance bottleneck in most cases.

If knowing this access time is not critical, and it certainly is not for a busy mirror server, it can be turned off by mounting the filesystem with the noatime option.

For XFS, the "logbufs" mount option allows the administrator to specify how many in-memory log buffers are used. We don't entirely understand what these log buffers do, however we increased the number to its maximum (8) and found that this increased performance. According to the mount(8) manpage, this performance increase comes at the expense of memory, however our favoured approach has been to treat memory as an expense in such situations.

As we also make our filesystems available over NFS, we used the "fs/xfs/refcache_size" sysctl provided by XFS to increase the amount of memory XFS would devote to caching references for NFS. We increased this number to 5120.

After nearly 3 years of use, some problems with XFS became noticeable. As the number of used inodes in the filesystem grows XFS becomes very slow at directory traversal. Although this did not impede web-serving it did have a heavy impact on how quickly we could synchronise content with the primary sources.

It could take rsync over 4 hours to traverse the Debian project hierarchy for example, by which time the TCP session opened by rsync had long-since timed out and the rsync had failed.

As such, we are migrating to using ext3 as our filesystem. Ext3 is still not quite as fast as XFS.

```
XFS:

Version  1.02b       ------Sequential Output------ --Sequential Input- --Random-
                     -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine        Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP  /sec %CP
cassandra       24G 25190  99 62029  21 30634  14 23159  97 61744  15 244.5   1
                     ------Sequential Create------ --------Random Create--------
                     -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
files:max:min         /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP
cassandra        16  4554  50 +++++ +++  3348  30  4270  49 +++++ +++  2998  32

Ext3:

Version 1.02b       ------Sequential Output------ --Sequential Input- --Random-
                     -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine        Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP  /sec %CP
cassandra       24G 22829  97 53363  35 29719  13 24764  96 63875  12 152.1   0
                     ------Sequential Create------ --------Random Create--------
                     -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
               files  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP
                  16 29579  98 +++++ +++ +++++ +++ 31836  96 +++++ +++ 32501  87

Ext3 with dir_index:

Version 1.02b       ------Sequential Output------ --Sequential Input- --Random-
                     -Per Chr- --Block-- -Rewrite- -Per Chr- --Block-- --Seeks--
Machine        Size K/sec %CP K/sec %CP K/sec %CP K/sec %CP K/sec %CP  /sec %CP
cassandra       24G 22479  97 53690  36 29763  14 23733  96 62687  13 147.7   0
                     ------Sequential Create------ --------Random Create--------
                     -Create-- --Read--- -Delete-- -Create-- --Read--- -Delete--
               files  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP  /sec %CP
                  16 32270  97 +++++ +++ +++++ +++ 31053  99 +++++ +++ 32164  92
```

Ext3's read performance is still acceptable. The ext3 dir_index option is an option whereby

ext3 uses hashed binary-trees to speed up lookup in directories. This has proved much faster for directory traversal and the Debian project can now be traversed in 3 minutes.

Our ext3 build commands were;

```
mkfs.ext3 -j -b 4096 -O dir_index /dev/sdc1
```

Although both ext3 and XFS allow the use of separate devices for storing journal information, we have chosen not to try this as we prefer to store the journal on the same volume to minimise problems during any potential recovery. It is likely that using a fast 15k RPM device for the journal information may speed up the filesystems.

ext3 does not have an equivalent of XFS's logbufs option, as such our mount options are merely;

```
/dev/sdc1       /ftp            ext3    rw,noatime  0 1
```

ext3 does have an interesting "data=writeback" mount option which changes the nature of the journaling to achieve higher throughput, however we have avoided this option due to the accompanying warning about file corruption following a crash.

4.2 NFS

The network filesystem, which retrieves content via the network rather than from physical disks directly, is a very different beast from other filesystems. At one stage we attempted to arrange ftp.heanet.ie as a front-end web/ftp server with a separate system running Linux as the back-end and content being retrieved via NFS.

To facilitate this we spent some effort maximising the throughput from our NFS configuration, in addition to the XFS-related sysctl mentioned above. For our NFS setup we had the luxury of creating our own private LAN for NFS traffic. As there were only two systems in the NFS network we chose to use a single ethernet cable for creating this LAN.

When connecting Gigabit interfaces, a crossover cable is not necessary (and would forc the network into 100Mbit degraded mode). Initially we used two on-board Broadcom interfaces however later changed to Intel due to the large MTU supported by Intel cards (16000 bytes Vs 9000 bytes). As you may have guessed by now, the intention of this LAN was to create a network which would sustain jumboframes.

By using jumboframes, it was possible to send more data in each NFS packet and thus reduce the amount of NFS operations drastically. We set the MTU on each side to 9000 bytes (and then later to 16000) and by specifying the rsize and wsize mount options on the NFS client could force NFS to use extra-large sizes when reading and writing packets.

Our NFS mount options were:

```
192.168.0.2:/ftp /ftp           nfs     rw,nolock,noatime,\
nfsvers=3,rsize=8192,wsize=8192,hard,intr     0       0
```

We chose NFS version 3, as it appeared to be the most stable version supported by Linux. Although the nolock mount option is usually inadvisable, in our situation only one system in the NFS setup is ever writing to the filesystem and it is safe to turn off locking.

As per the Linux NFS HOWTO [TBM02], we increased the amount of memory available to the TCP/IP input queue on the NFS client;

```
sys/net/core/rmem_default=262144
sys/net/core/rmem_max=262144
```

Our NFS server options (in /etc/exports) were relatively standard:

```
/ftp   192.168.0.1(rw,secure,async,no_subtree_check)
```

By utilising all of these options it was possible to get bonnie++ metrics within 1% of raw access to the native filesystem being served.

This however led us to be overconfident about Linux's NFS abilities. Our experiment with using a backend Linux NFS server only lasted a few days. As it turns out, Linux's NFS threads are in-kernel processes and as such afforded a lower scheduling priority than userland processes.

Although we wanted the kernel to prioritise the NFS, if any userland processes were run these would impact the NFS performance heavily - leading to deadlock as the NFS processes scrambled to catch up only to find themselves being afforded less and less time.

In our experience, Linux scaled to approximately 4,000 concurrent NFS operations. This however was not sufficient for our needs.

4.3 Choosing a kernel

There are very many Linux kernel options, modules, versions and patch-sets to choose from - far too many to test individually and determine their usefulness empirically. To a large extent decisions about which kernel to use are made in a broader way. It is easy to compare 2.4 Vs 2.6, less easy to then decide which specific features to enable and use.

The new ftp.heanet.ie ran initially on the SGI Linux 2.4 kernel which included the XFS patch-set. This kernel proved sufficient for our needs however when the 2.6 kernel was released - and included XFS code - we decided to give it a try.

The scheduler in the 2.6 kernel appears to have markedly improved. Previously we had thought that about 12,000 sessions was our maximum, however after simply upgrading to the 2.6 kernel we were hitting the compiled-in 20,000 limit of Apache without any additional effort.

2.6 was not without its problems however. Until 2.6.10 the memory manager proved very temperamental and hard to control, frequently killing seemingly arbitrary userland processes.

In terms of kernel features, we have experimented with preempt-ability. Although it is somewhat unusual to use preempt-ability on a server we decided that it would be interesting to determine its effect - however this proved hard to judge, though did not seem to have an negative effect.

One unusual feature we enable is connection tracking. As our CPU load is quite small we enable connection tracking to monitor the TCP and UDP sessions to ftp.heanet.ie. With a large volume of users there are occasional TCP/IP bugs and malicious activities that can be spotted with this. On a daily basis there are also several thousand instances of;

```
TCP: Treason uncloaked! Peer 81.15.218.99:11319/80
shrinks window 968215270:968219614. Repaired.
```

We do ensure that connection tracking does not limit how many connections we can handle;

```
net/ipv4/ip_conntrack_max=6553600
```

4.4 Tuning the Kernel

One of the most important tunables for a large-scale webserver is the maximum number of file descriptors the system is allowed to have open at once. The default; a painfully small 50498, is not sufficient when serving thousands of clients. It is important to remember that regular files, sockets, pipes and the standard streams for every running process are all classed as file-descriptors and that it is easy to run out. Thus we added two zeroes to our limit;

```
fs/file-max=5049800
```

There are two main components of the kernel that are very much worth tuning, the virtual memory manager and the networking stack.

4.4.1 Tuning the VM

The Linux VM is fairly complicated and provides a large number of options that can be tuned. I recommend reading "Understanding the Linux Virtual Memory Manager"[Gor04] by Mel Gorman for an in-depth guide.

In Linux, the VM does not just manage the memory allocated to processes and the kernel but also manages the in-memory cache of files we encountered earlier. By far the easiest way to "tune" the VM in this regard is to increase the amount of memory available to it. This is probably the most reliable and easy way of speeding up a webserver - add as much memory as you can afford to.

The aim should be that as much as possible of the most popular content is served from memory, but for the sake of system stability we also want to ensure that there is enough memory for the processes that are running.

The first thing to realise in this regard is that the VM takes a similar approach to mod_disk_cache for freeing up space - it assigns programs memory as they request it and then periodically prunes back what can be made free (this includes cached file data). If a lot of files are being read very quickly, the rate of increase of memory usage will be very high.

If this rate is so high that memory is exhausted before the VM had had a chance to free any there will be severe system instability. To correct for this we have set 5 sysctl's:

```
vm/min_free_kbytes = 204800
vm/lower_zone_protection = 1024
vm/page-cluster = 20
vm/swappiness = 200
vm/vm_vfs_scan_ratio = 2
```

The first of these sysctl's establishes that we would like the VM to aim for at least 200 Megabytes of memory to be free. This may seem like a lot, but when serving a lot of content that amount of data can be added to the cache very rapidly. With one particular version of the 2.6 kernel we had to set this sysctl to aim for 2 Gigabytes of memory to be kept free, however since then the algorithm for freeing memory appears to have improved.

The second sysctl establishes the amount of "lower zone" memory, which is directly addressable by the CPU, that should be kept free. It is important to keep memory in this zone free as Linux pages memory into this zone for a large number of memory operations.

The third sysctl, "vm/page-cluster" tells Linux how many pages to free at a time when freeing pages. 20 is a rather large value but when dealing with 12gb of ram it is sometimes necessary to take the sledgehammer approach in order to free it quickly enough.

The fourth sysctl, "swappiness" is a very vague and ill-defined sysctl which seems to boil down to how much Linux "prefers" swap, or how "swappy" it should be. We set this number very high (the default is high) as we like to ensure that the system will prefer swap in any situation where it might otherwise lead to an out-of-memory problem.

And lastly, the "vm_vfs_scan_ratio" sysctl sets what proportion of the filesystem-data caches should be scanned when freeing memory. By setting this to 20 we mean that 1/20th of them should be scanned - this means that some cached data is kept longer than it otherwise would, leading to increased opportunity for re-use.

4.4.2 Tuning the Networking Stack

In addition to the connection tracking sysctl mentioned above, we currently use 6 other sysctl's:

```
net/ipv4/tcp_rfc1337=1
```

```
net/ipv4/tcp_syncookies=1
net/ipv4/tcp_keepalive_time = 300
net/ipv4/tcp_max_orphans=1000
sys/net/core/rmem_default=262144
sys/net/core/rmem_max=262144
```

We enable TCP syncookies and the RFC1337 options for security reasons, protecting against SYN floods and TIME WAIT assassination respectively.

We lower the default tcp keepalive time (two hours) to 5 minutes to avoid the situation where httpd children handling connections which have not been responsive for 5 minutes are not needlessly waiting in the queue. This has the small impact that if the client does try to continue with the TCP session at a later time it will disconnect.

The max_orphans option ensures that even despite the 5 minute timeout there are never more than 1,000 processes held in such a state, and will instead start closing the sockets of the longest-waiting processes. This prevents process starvation due to many broken connections.

The final two options we have seen before with NFS and increase the amount of memory generally available to the networking stack for queueing packets.

4.4.3 Pluggable I/O schedulers

Recently the option of pluggable/selectable I/O schedulers has become available in the Linux kernel. We have not yet experimented with changing our I/O scheduler from the default, however this may be an interesting area and should be of interest to anyone building a large-scale webserver.

4.5 Hyperthreading

The author at least has almost no understanding of hyperthreading other than that it is a technology which makes one processor show up as two with the aim of improving resource usage efficiency within the processor. Many debates have taken place on whether it is beneficial or not. Rather than try and follow these discussions our approach was to benchmark the webserver under two conditions; hyperthreading enabled and hyperthreading disabled.

Hyperthreading enabled resulted in a 37% performance increase for our load (from 721 requests per second to 989 requests per second, with the same test). We have therefore left it enabled.

5 System Design

> "A common mistake people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools"
>
> Douglas Adams

Coming in to 2002, ftp.heanet.ie was running Apache 1.3.x, with Digital Unix 3 as the Operating System. The hardware was a 200Mhz Alphaserver with a vast 20 Gigabytes of storage and a princely 10Mbit/sec (half duplex) network connection. Debates over whether the network or the CPU would have given away first under any real load will forever remain speculative, however the box did a great job considering it also ran a Listserv service with several hundred mailing lists and about 30,000 mails per day, along with tftp and syslog services.

In March of 2002, it was decided to migrate all of the services on Avoca - the old system - to the anagrammatic Athene, a brand new Alphaserver DS20E which boasted a 667Mhz CPU, 1.5Gb of memory, 30Gb of storage and the possibility of gigabit connectivity. In practise, only the 100Mbit/sec (but now full duplex) connectivity was used while ftp.heanet.ie was hosted on this machine.

But it was in the summer of 2002 when HEAnet decided to build a dedicated mirror server that the question of system design came into the equation.

5.1 Hardware and Operating System choice

Both the Operating System and hardware vendor were relatively self-selecting. At the time, HEAnet was undergoing a rapid expansion into services and a complete reorganisation of its existing services network. As part of this, rack-mount Dell Poweredge systems had been chosen

early on as the servers of choice. The benefits of having an on-site cold-spare system and a favourable relationship with Dell outweighed any concerns we had about using a non-64-bit platform.

By this stage we were comfortable with the platform and had a chance to determine whether it would meet our initial requirements. Similarly, HEAnet had chosen Debian as it's standard server operating system.

For more details on HEAnet's choice of Operating System, hardware platform and much more detail surrounding their design, see our "Case-study on best-practise for Operating System Management" paper[MW03].

5.2 Canyonero.heanet.ie

The first system of the re-launched ftp.heanet.ie was canyonero, a Dell 2650, with 2 1.8Ghz P4 Xeon processor, initially 2Gb of memory, quickly upgraded to 4Gb of memory. As with all of our systems, there were two 36Gb system-disks in a (software) RAID-1 container.

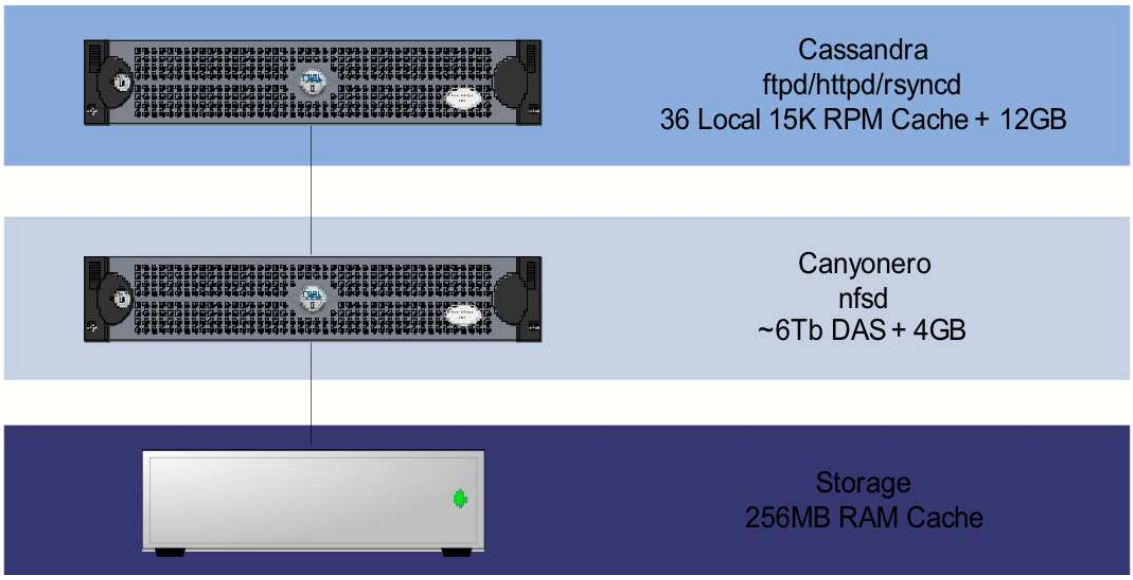
Initially storage was provided by a single Fibrenetix Zero-D RAID box. Taking 16 off-the-shelf 160Gb IDE disks, the Zero-D then provides a ultra-320 SCSI interface to the host.

Although the RAID5 configuration was capable of providing 2.2 Terabytes of storage, unfortunately x86 32-bit systems are only capable of addressing 2 Terabytes of contiguous disk. Rather than split the disks into multiple RAIDs, we chose to build a slightly smaller array of just over 2 Terabytes in size and have an additional hot-spare disk in the RAID.

One year after re-launching the service, a second Fibrenetix system was purchased with 16 250Gb IDE disks, providing an additional 4 Terabytes of storage. In this case the storage was split into two RAID5 containers, each 1.6 Terabytes in size (the remainder being taken up by parity and spare disks).

Canyonero served as an extremely good mirror server. It's peak-usage in production was a spike of 480Mbit/sec with approximately 11,000 concurrent downloads.

5.3 Attempted Multi-system architecture

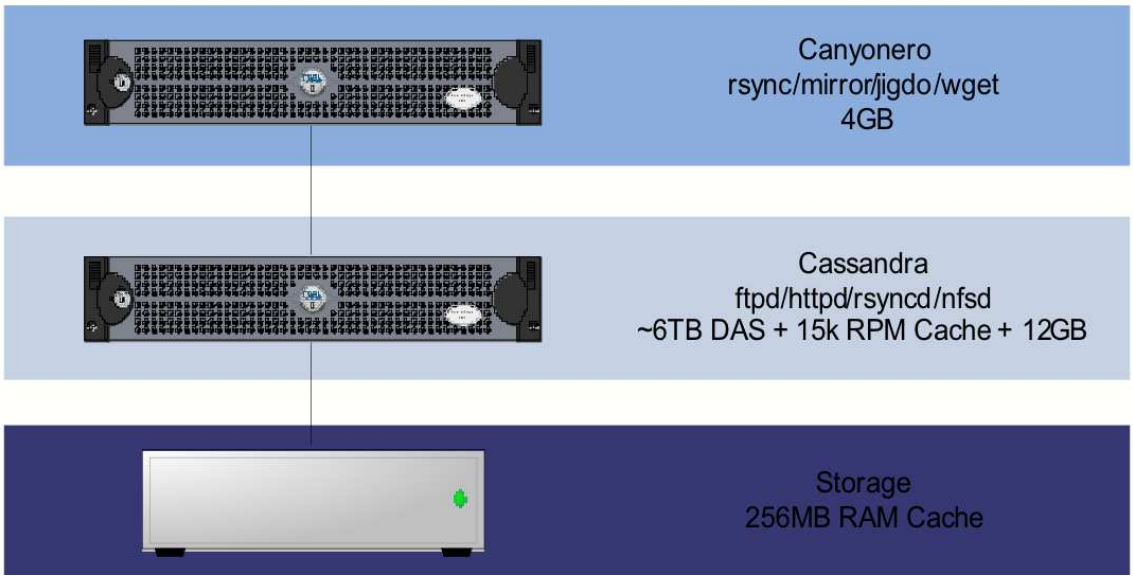


After a year or so of operation, we decided it would be desirable to have more scalability. To that end, we purchased Cassandra.heanet.ie, a Dell 2650, with 2 2.4 Ghz Xeon processors, 12Gb of memory and the usual 2 system disks. We also purchased the 15k RPM SCSI disks to experiment with mod_disk_cache.

The aim was to leave the disk arrays connected to Canyonero but to configure Canyonero as an NFS server, with Cassandra accessing the filesystems as an NFS client and serving the content to end users. Future scalability would have been achieved through expanding the number of front-end servers horizontally with mod_disk_cache minimising the volume of content which would need to be retrieved from the back-end.

As described earlier however this setup ultimately failed due to the Linux NFS implementation limitations and lasted less than one week in production.

5.4 Cassandra.heanet.ie

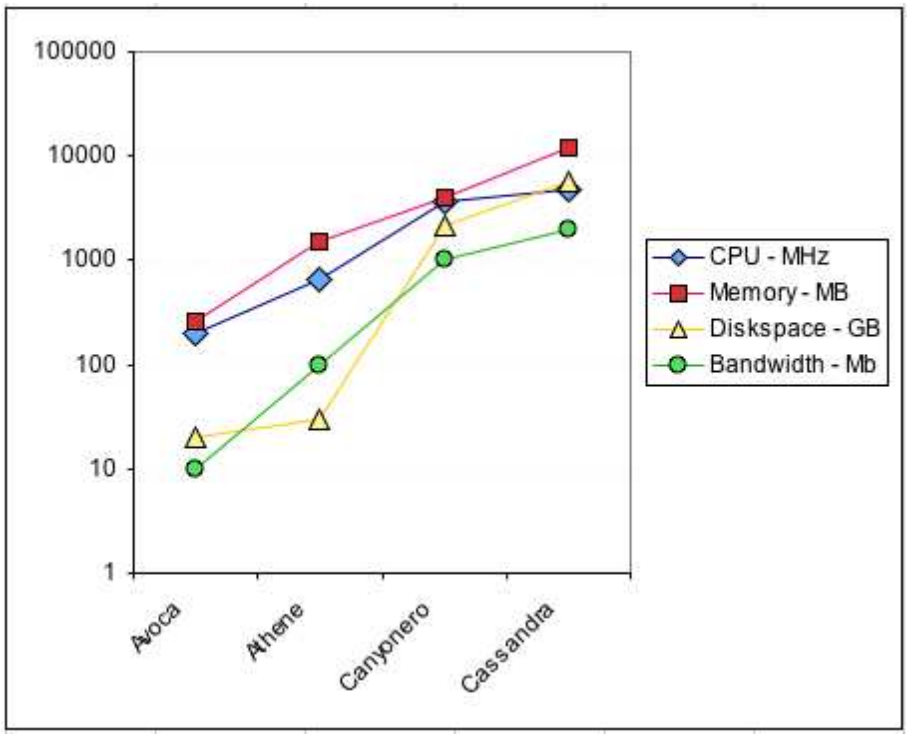


To cope with the architecture failure, we reversed the roles of canyonero and Cassandra somewhat and connected the storage directly to Cassandra. Apache was left running on cassandra and did not need to use NFS to access content, however Cassandra was also now configured as an NFS server.

The mirroring scripts which synchronise content from its primary locations were then migrated to Canyonero to relieve cassandra of some of the more CPU intensive tasks. Linux has been handling this much lighter NFS load with no major problems so far, though we do get reports of file corruption possibly caused by NFS errors (usually fixed by the synchronisation scripts themselves).

This remains the current architecture of ftp.heanet.ie.

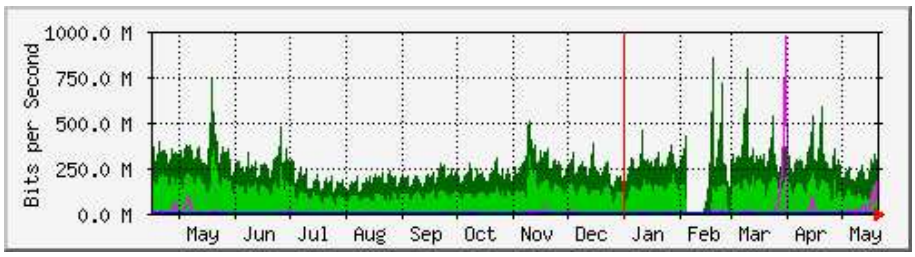
5.5 Summary: Time-line of ftp.heanet.ie



For future planning and comparison purposes, we find it useful to take a look at ftp.heanet.ie's progression over time. Note that the above graph is on a logarithmic scale. As the bandwidth available to the average internet user increases it takes exponentially more resources to serve them.

However to some extent greater bandwidth is mitigating some of the scheduler-type problems. For the most part, the rate at which we can ship content, and therefore the amount of time processes must spend idle in the queue, is ruled by our users' bandwidth.

Although we are shipping much more content and serving many more requests per day, our average level of concurrency has not changed much in 2 years. As the amount of requests increases, so does the rate at which they may be served. We're not certain if this will hold however.



6 Future changes for ftp.heanet.ie

> "Prediction is very difficult, especially of the future.'
>
> Niels Bohr

6.1 Jumboframes

Jumboframes, which allow for ethernet frames above the usual 1500 bytes in size, are in the process of being deployed at HEAnet. With the services and backbone networks both being configured to operate up to 9000 byte frame-sizes, we expect ftp.heanet.ie to be capable of sending such packets by the time you read this!

Hopefully Jumboframes will prove somewhat useful in tackling the TCP bottlenecks caused by round-trip-time and allow us to ship content faster to our European academic peers and users of Internet2.

6.2 Multicast services

Somewhat unusually for a mirror service, we hope to offer some multicast services in the near future. Although there are protocols such as FLUTE [TPRW04] which can be used to ship large amounts of content via multicast, we initially intend to host only a multicast notification service.

ftp.heanet.ie is host to a large amount of time-sensitive data, such as virus signatures, anti-spam rulesets and security-updates. Multicast is an efficient way of making clients aware of when these updates are available. To that end, HEAnet is currently developing and trialling a Multicast Event Notification Daemon.

6.3 mod_ftpd?

One possible future architecture for HEAnet is to migrate to a cluster of caching reverse-proxies, using HTTP for content retrieval from the back-end. However there are currently no FTP servers which will act as HTTP reverse proxies.

One option which is being examined is to use mod_ftpd within Apache to serve FTP content. mod_ftpd does not currently work with IPv6 or mod_cache however, though HEAnet is doing some work towards adding this support.

6.4 64-bit and FreeBSD

One of the more tentative plans for ftp.heanet.ie is a migration to FreeBSD running on a 64-bit platform (more than likely AMD64). In addition to the benefits of being able to address large

volumes of disk space and memory directly, we believe a migration to FreeBSD is worthwhile in order to experiment with its memory management and process scheduling. Using FreeBSD may also solve our long-standing sendfile problem, due to its differing implementation of the sendfile call.

References

[Gor04] Mel Gorman. Understanding the linux virtual memory manager. April 2004.

[MJ] David Mosberger and Tai Jin. httperf - a tool for measuring web server performance. `http://www.hpl.hp.com/personal/David_Mosberger/httperf.html`.

[MW03] Colm MacCarthaigh and Colin Whittaker. A case-study in best practise for operating system management. February 2003. `http://www.sage-ie.org/papers/case_study/`.

[spe99] spec.org. Specweb99 benchmark. 1999. `http://www.spec.org/osg/web99/`.

[TBM02] Seth Vidal Tavis Barr, Nicolai Langfeldt and Tom McNeal. Linux nfs-howto. August 2002. `http://nfs.sourceforge.net/nfs-howto/`.

[TPRW04] R. Lehtonen T. Paila, M. Luby, V. Roca, and R. Walsh. Rfc3926: Flute - file delivery over unidirectional transport. October 2004. `ftp://ftp.rfc-editor.org/in-notes/rfc3926.txt`.

Acknowledgements

The author is eternally grateful to Noirin Plunkett, who reviewed and corrected this paper's many spelling mistakes and provided the htcacheclean and ftp.heanet.ie-over-time graphs. Brian Scanlan, HEAnet, created the autobench graphs used in this paper and Colin Whittaker served as technical reviewer.