CFTP – A Caching FTP Server

Mark Russell and Tim Hopkins
Computing Laboratory
University of Kent
Canterbury, CT2 7NF
Kent, UK

August 5, 1998

Abstract

By analyzing the log files generated by the UK National Web Cache and by a number of origin FTP sites we provide evidence that an FTP proxy cache with knowledge of local (national) mirror sites could significantly reduce the amount of data that needs to be transferred across already overused networks.

We then describe the design and implementation of CFTP, a caching FTP server, and report on its usage over the first 10 months of its deployment. Finally we discuss a number of ways in which the software could be further enhanced to improve both its efficiency and its usability.
KEYWORDS: mirroring, caching, log summation, usage statistics, object oriented.

1 Introduction

International bandwidth is an expensive and scarce resource which is often wasted by inefficient patterns of use. Economic constraints mean that the academic community, at least, cannot continue to increase capacity in order to satisfy an ever growing demand for data transfer.

Of the three dominant protocols used on the network, HTTP [4], FTP [9] and NNTP [7], the first two offer scope for saving bandwidth because they are used primarily for read-only access to relatively static bodies of information. One way of achieving this is to keep local copies of resources as they are requested and to use this local data rather than re-requesting the original from the remote site. This approach is already being used by existing systems, including numerous FTP mirror sites (for example, HENSA Unix at the University of Kent [1]) and various Web caching systems (for example, the Netscape proxy [3] and the Squid cache [10]). The approach reported in this paper is to combine FTP caching with FTP mirroring.

In section 2 we discuss the analysis of a large quantity of log data collected from a number of sources, and draw some conclusions regarding both the way in which international bandwidth is being utilized by the academic community and what opportunities exist for reducing the number of times the same data is requested from the origin site. We would expect that the conclusions we have drawn would be applicable to a much wider user community and that the approach that we have taken would offer similar improvements to any site which is close to capacity in its use of available international bandwidth.

The design and implementation of a caching FTP server are discussed in section 3 and preliminary results from the first 10 months of its use are presented in section 4.

Finally in section 5 we consider a number of ways in which the existing software could be improved in both efficiency and usability.

Date	http Requests	http %Total	FTP Requests	FTP %Total	other Requests	other %Total	Total Requests
1st –7th	1103838	96.1%	23692	2.1%	21553	1.9%	1149083
8th –14th	1207191	95.4%	33037	2.6%	25599	2.0%	1265827
15th–21st	1405571	95.8%	33794	2.3%	28558	1.9%	1467923
22nd–28th	1336244	95.6%	36675	2.6%	24831	1.8%	1397750
Total	5052844	95.7%	127198	2.4%	100541	1.9%	5280583

Table 1: Requests to cache server by protocol

2 Statistics Gathering

We wished to obtain data on the existing use of international bandwidth, for example, how much use was being made of existing caching and mirroring facilities, how many sites were bypassing UK caches, and how the load was distributed across the origin FTP servers. We first turned to the access logs maintained by the UK Academic National Web Cache which at the time was located at HENSA Unix. This service allowed user requests for Web pages to be directed to a number of cache machines situated at a central site. If a requested page had been asked for recently by another user the page retrieved earlier was returned. Otherwise the page was obtained from the origin site and forwarded to the requestor; a copy was also kept at the cache for possible future use. The cache machines had access to a dedicated line to the United States in order to speed up requests to the origin site. Many users request FTP URLs from their browser so this information provided the closest thing we had to a global picture of FTP access patterns within the UK academic community.

This data has some biases, as it only covers users and sites that have configured their browsers to use the national cache and, for that subset of *.ac.uk* sites, it only covered browser initiated FTP traffic. However we believe that it is probably representative of the whole community and almost certainly representable of the part of the community that could be persuaded to use a new FTP caching facility.

Probably the most serious possible bias was that the data only covered FTP traffic initiated via a Web browser: the worry being that this traffic would differ from that initiated by plain FTP clients. Analysis of the HENSA Unix FTP logs points to a growing fraction of accesses via Web browsers (i.e., using HTML rather than FTP). For accesses by *.ac.uk* sites during 1997, 36% of the requests and 30% of the data shipped from HENSA Unix were via HTML.

The conclusion we drew from this was that users find it more convenient to use their Web browsers to explore FTP sites than to utilize the traditional text based FTP interfaces.

Tables 1 and 2 show the breakdown, by weeks, in traffic by requests and the amount of data shipped respectively for one of the six National Web Cache hosts during the four week period Monday 1st to Sunday 28th May 1995 inclusive. The results are grouped by the protocol specified in the URL and the column labelled *other* covers URL types such as gopher, Wais, file, etc. The figures given are for successful requests only.

Not surprisingly for a Web cache, 95% of the successful requests use HTTP and only 2.4% use FTP. However, the proportion by volume of the data was around 13% and indicated that FTP items are, on average, around five times larger than HTML documents.

A breakdown of the FTP traffic by origin site showed that although a mere seven

	http		FTP		other		Total
Date	Data(MB)	%Total	Data(MB)	%Total	Data(MB)	%Total	Data(MB)
1st –7th	13645	86.3%	1749	11.1%	412	2.6%	15808
8th –14th	15539	84.3%	2425	13.2%	468	2.5%	18434
15th–21st	17384	85.0%	2548	12.5%	519	2.5%	20451
22nd–28th	16225	83.0%	2830	14.5%	494	2.5%	19550
Total	62795	84.6%	9553	12.9%	1895	2.6%	74244

Table 2: Requests to cache server by protocol

#Sites	%Sites	%Requests	%Data
4	0.1	40.0	41.8
7	0.2	46.2	50.5
16	0.5	54.4	56.5
31	1.0	61.8	62.6
60	2.0	70.0	69.0
148	5.0	80.1	78.7
294	10.0	86.9	85.6
1464	50.0	97.9	98.2
2926	100.0	100.0	100.0

Table 3: Distribution of requests by site

sites accounted for just over 50% of the volume requested there was a very large tail to the distribution with the last 1464 sites accounting for only 2%. Details of the distribution are given in Table 3.

We then looked in more detail at the distribution of requests at some of these sites as we wished to ascertain whether requests were spread evenly over the filestore, or were concentrated on a few heavily used files and directories. Table 4 shows, for a selection of servers, the total number of requests to the server and the amount of data shipped, followed by the number of distinct documents requested. The last three columns show the proportion of the total data shipped by the site accounted for by the most popular (in terms of of volume of data accounted for by requests for a document) 10, 20 and 100 documents. The sample includes both heavily and lightly used sites.

Table 4 shows very clearly that the load tends to concentrate on a few heavily used documents. For example, on *ftp.netcom.com*, just 20 documents accounted for a quarter of all the data shipped. This data suggests strongly that caching will be more effective than mirroring and, indeed, that mirroring some of these sites would result in much wasted bandwidth due to fetching items that will never be accessed.

In addition we attempted to produce a more complete picture of FTP access patterns by obtaining data on accesses by the UK academic community to a number of overseas FTP sites. We obtained long term data from two large US FTP servers, *ftp.uu.net* and *sunsite.unc.edu*, and snapshot summaries from 30 others.

Table 5 provides an overall summary of the long term data and Table 6 shows how much of the traffic was generated by the UK mirror sites at HENSA and Imperial College. Together the mirror sites accounted for 53% of the data shipped to hosts under *.ac.uk* from *sunsite.unc.edu* and 90% of that from *ftp.uu.net*.

An analysis of the most popular 10, 20, 100 and 1000 documents is given in Table 7. For direct transfers, i.e., those not going to a mirror, the top 10 of the

Host	Total Requests	Total data (Mb)	Docs	Data due to top N docs 10	20	100
ftp.netcom.com	21400	859.6	1143	16%	25%	65%
ftp.sunet.se	16691	1368.6	4668	16%	21%	38%
ftp.tu-muenchen.de	9195	1217.6	697	20%	31%	74%
ftp.univ-lille1.fr	2333	237.1	906	32%	38%	59%
ftp.tcp.com	792	29.6	231	52%	65%	94%
mirrors.aol.com	578	8.4	408	60%	76%	92%
ftp.best.com	321	14.9	135	46%	63%	100%
ftp.borland.com	199	8.8	161	71%	81%	99%
louie.udel.edu	136	0.8	22	93%	100%	100%
solaria.mit.edu	114	4.7	39	74%	91%	100%
gatekeeper.dec.com	102	10.4	45	94%	99%	100%

Table 4: Distribution of document requests

Site	Start	End	Requests	Data (MB)
ftp.uu.net	12 Aug 1995	16 Feb 1996	129503	10162
sunsite.unc.edu	3 Nov 1995	15 Oct 1996	247272	34030

Table 5: FTP long term log data

7000+ documents retrieved from *ftp.uu.net* accounted for 30% of the data shipped. The access figures for the mirror sites are surprising: just 10 documents account for almost half the data transferred. We had supposed that the mirror sites would have shown a far more even access pattern given that they are normally retrieving any documents that have changed. A closer look at what was being retrieved revealed that nine of the ten files were index files!

The problem here is that the origin FTP servers generate new index files, often in several formats, usually on a daily basis. Even when compressed these files may be several megabytes in size. Since users rarely request index files they should probably not be mirrored, although this kind of selective filtering is additional administrative work for the mirror sites.

Finally, Table 8 shows summary information of the 22 sites from which we obtained snapshot data and which had more than 10 requests/day from *ac.uk* sites.

As can be seen from the table there was a wide range of usage levels (1Mb to 1Gb per day) and a wide range of time periods (2 to 295 days). We were dependent on what logs each server administrator had available and we have assumed that the access patterns did not change radically over the period covered. The *Cached* column in Table 8 records the percentage of requests that came via a Web proxy

Host	sunsite.unc.edu Requests		sunsite.unc.edu Data (MB)		ftp.uu.net Requests		ftp.uu.net Data (MB)	
hensa.ac.uk	40610	16%	9224	27%	99873	77%	8437	83%
ic.ac.uk	61745	24%	9128	26%	15691	12%	743	7%
Others	144917	60%	15678	47%	13939	11%	982	10%

Table 6: Mirrored vs non-mirrored data

Host	Total Requests	Total Data (MB)	Docs	%Data due to top N docs 10	20	100	1000
uunet direct	14147	1004	7037	30%	41%	62%	92%
uunet to mirrors	115356	9158	25955	47%	54%	63%	79%
sunsite direct	101000	16344	27066	5%	7%	19%	61%
sunsite to mirrors	146272	17686	45656	8%	11%	22%	57%

Table 7: Document access patterns

server. This was calculated by totaling the requests from the HENSA Unix proxy server hosts.

The table shows that there is still much scope for improved caching and mirroring of FTP servers as, for most of the servers providing data, 90% or more of the traffic was via direct connections.

We were thus able to draw the following conclusions

1. Traffic is spread across a large number of FTP servers. Although a large percentage of the traffic is concentrated on a small number of servers, a significant fraction of the load is spread across a large number of small to medium servers (see Table 3).

2. Traffic to each server tends to be concentrated on a very small number of files, frequently with the top 20 most popular files accounting for 30% or more of the data shipped (see Table 4).

3. For some large sites, much of the load from the *.ac.uk* domain is handled by mirrors although there is still a significant amount of direct access (see Table 6).

4. Mirrors can waste bandwidth by repeatedly pulling index files although such transfers are usually performed during the overnight slack period on the transatlantic connection (see Table 7).

5. There is scope for improved caching and/or mirroring as many small to medium sites are still getting a large proportion of their traffic from direct client connections.

3 Overview of Design and Implementation

3.1 Design Constraints

From the above analysis it seemed clear that some form of caching FTP server (CFTP) would be effective. The FTP traffic was distributed across far more servers than could be practically mirrored and a cache would automatically duplicate the FTP resources as necessary. However we also wanted to take advantage of the large amount of material that was already being mirrored by sites in the UK; it was thus clear that the optimal solution would combine caching and mirroring.

Because of user reluctance at installing new software and the difficulty of distributing new clients we wanted to use existing client software; this meant using either traditional FTP clients or Web browsers as the interface. We thus knew from the outset that multiple interfaces to the system would be necessary, which meant that the underlying functionality had to be available as a library of software, usable by several different server applications.

Host	1995 From	To	Days	Requests per day	Data (MB) per day	Cached
archive.uwp.edu	11 Nov	16 Nov	5.5	409	32.66	5.6%
cdt.luth.se	8 Nov	9 Nov	1.9	3282	1035.79	4.9%
cs.ruu.nl	31 Oct	17 Nov	16.6	256	13.23	9.9%
dsi.unimi.it	11 Jul	18 Nov	130.0	15	2.79	5.3%
garbo.uwasa.fi	15 Nov	17 Nov	2.0	431	55.72	13.5%
granite.mv.net	31 Oct	15 Nov	15.0	15	1.09	4.0%
ifi.uio.no	8 Sep	16 Nov	69.7	62	10.05	6.5%
inf.tu-dresden.de	30 Aug	19 Nov	81.8	43	11.23	12.1%
info2.rus.uni-stuttgart.de	9 Nov	19 Nov	9.3	1046	68.53	18.4%
lysator.liu.se	29 Oct	16 Nov	17.9	2042	49.25	17.1%
math.uni-hamburg.de	1 Jan	16 Nov	295.7	20	3.43	9.6%
mathworks.com	2 Oct	1 Nov	29.7	179	9.77	27.9%
nvg.unit.no	15 Nov	18 Nov	2.1	790	44.63	14.3%
oak.oakland.edu	7 Nov	16 Nov	9.1	71	7.80	6.2%
papa.indstate.edu	1 Oct	16 Nov	46.7	31	3.95	10.2%
pascal.zedat.fu-berlin.de	30 Apr	19 Nov	202.2	69	11.69	7.7%
povray.org	1 Nov	24 Nov	23.1	248	52.75	6.1%
rhrk.uni-kl.de	1 Nov	17 Nov	16.5	95	9.44	7.6%
sdi.slu.se	4 Nov	11 Nov	7.0	2691	416.21	6.9%
team17.com	7 Sep	16 Nov	70.2	17	10.78	9.3%
utexas.edu	25 Oct	16 Nov	22.1	12	2.81	4.0%
voa.gov	5 Nov	16 Nov	12.0	47	7.77	0.0%

Table 8: Log summary data

Another force driving the design was the need to support various different methods of fetching resources. A given object might be fetched from an origin FTP server, from local filestore, from a mirror server or from a previously cached copy. We might also, later on, want to handle protocols other than FTP. This argued for splitting the various sources into well-separated modules, with a common interface to the core system.

Finally, the system needed to be highly configurable. We wanted to be able to experiment with, for example, various different ways of combining caching and mirroring, without embedding these decisions in the code. We therefore aimed to move as many policy decisions as possible out of code and into configuration files.

3.2 Major Decisions

The major design decision was that the core of the system should be structured as a virtual filestore tree. The virtual tree is initially empty, and is populated by mounting “filesystems” on paths. This process is similar to the way filesystems are mounted on a typical UNIX system. A configuration file specifies what to mount where, and thus determines the layout of the tree that the client sees.

Although our virtual tree is analogous to a UNIX filestore tree, there is an important difference. All the path resolution and handling of mount points is within a single process. Since there are no system calls involved in crossing mount point boundaries, this means that mount points are cheap, which is important because we compose them to build up combinations of functionality. Each “filesystem” type implements a single abstraction, such as caching, FTP access or local filestore access.

The motivation for this approach was to partition the functionality into separate and isolated modules, and to insulate the client software completely from the underlying file access code. Although there were a number of problems along the way this approach has worked out well in practice.

We decided to implement our own FTP client software, rather than, say, driving an existing FTP client by sending commands down a pipe. We felt that having a separate FTP client process would cause problems with error handling, and that we would end up writing almost as much code to drive a separate process as to drive the FTP protocol directly. In practice implementing the FTP server and client protocol drivers was a small fraction of the overall work.

Some things were not anticipated in the original design, and only emerged as the result of implementation experience. For example, we initially had the server processes making direct connections to origin FTP servers, and only later discovered the need for a separate FTP manager process.

3.3 Implementation Language and Tools

Apart from a few administrative shell and perl scripts, the system is implemented in C++. This was a natural choice of language, and the system is inherently object oriented. For example, the notion of the various filesystem types, each being a different implementation of a common interface, has an obvious implementation as a C++ abstract class with derived classes for each filesystem.

To ensure maximum portability of the resultant software the code was written, as far as possible, to comply with relevant standards: POSIX.1 [6] for local operating system services, RFCs for the HTTP [4] and FTP [9] protocol, and the draft C++ standard for the language itself.

The decision to explicitly restrict the code to use only POSIX.1 OS facilities proved to be a good one. In particular it dealt with the problem of namespace pollution by header files. POSIX mandates that headers do not introduce extra symbols if the macro `_POSIX_SOURCE` is defined, and all the code is compiled in this mode. This eliminates a whole class of trivial but irritating problems caused by differences between header files under various versions of UNIX.

It was not possible to stay completely within the POSIX standard. For example, non-POSIX facilities were needed to handle symbolic links, fractional times, file truncation and some non-standard signals. But access to these facilities was isolated in a single (and small) `NonPosix` class, which alone was compiled without `_POSIX_SOURCE` defined.

In fact, and contrary to our previous experience with C, the language itself was more of a portability problem than OS facilities. Although C++ has been in widespread use for well over a decade, the language is still evolving. The forthcoming standard will make significant changes to the language, and the various compilers were tracking the standard at different rates. This meant that language features had different levels of portability, varying from universally implemented (most of the core of the language), through patchy availability (namespaces, exceptions) to almost non-existence (member templates, run time type identification).

Our strategy was to stay as close as possible to the draft standard. Our main compiler was the vendor's IRIX C++ compiler (based on the Edison Design Group front end). We also built the system using the Sun C++ compiler. Unfortunately we could not use g++ from the Free Software Foundation because it did not have adequate support for templates.

Workarounds were possible for some things in the standard that were not implemented by our compilers. A simple example was the new builtin `bool` type. We simply mapped this to `int` in a header file, and used it throughout the code. Obviously we did not get the additional type checking, but when this feature becomes available the code will be ready. Similarly, we used macros for the new casting syntax defined in the standard (`static_cast`, `dynamic_cast` etc). These all mapped to the old-style cast, but by using the new syntax to document which kind of cast was intended will allow an easy move to the new casts when they are supported.

The draft standard includes extensive additions to the standard library and we used standard library features where possible. Fortunately implementations of many of the major new features of the library are publically available. This code is currently part of the source of the system, but as versions of the standard library become available we will migrate to them.

The major standard C++ facility not currently supported by our compilers was exception handling. This was hard to work around, as the system involves extensive error handling. We used a textual message based error reporting scheme instead, although the code was designed in an exception-safe style. Functions were designed so that all resource cleanup is dealt with by destructors, thus ensuring that a return can be performed at any point without leaking resources.

Despite the problems of portability, C++ has been an effective language in which to both design and implement. The strong static type checking meant that code that compiled was likely to work first time, and that problems caused by changes were caught at compile time rather than run time.

Structuring the system into relatively independent objects had major benefits, notable among them being flexibility in the way objects were composed to form the system. For example, FTP connections were originally handled directly, and only later were moved into an FTP manager process. This change was easy because the code was encapsulated into an `FtpConnection` object, which did not know or care about its surrounding context.

Another key benefit was the support for implementing small concrete types. For example, filesystem paths are widely used in the system. These are represented as a `Path` class, rather then being passed around as strings. Code that uses paths is thus written in terms of logical operations on `Path` objects rather than using string manipulation; this makes a surprising difference to the readability of the code.

3.4 Implementation History

Our implementation strategy throughout the project was to build a series of working systems, each one adding to the functionality of the previous. This meant that we had a working (if minimal) system quite early on which gave us confidence that the design was workable. Although we did have to make some design changes as we proceeded, the object-oriented approach meant that this work consisted mainly of changing the relationship between objects rather than rewriting large sections of code.

Here is very brief list of the versions of the system that we implemented:

1. The virtual filestore, with support for local (POSIX) filesystems only, and accessed via the command line on the local host.

2. Access to FTP servers via the virtual filestore.

3. An FTP server, giving access to the virtual filestore via standard FTP client programs. At this point it was possible to connect to the system with an FTP client, and access external FTP servers, but only ones that were explicitly mentioned in the CFTP configuration file.

4. Support for "mounting" FTP servers on demand, thus allowing clients to access FTP servers of their own choosing.

5. Support for caching files and directory listings. At this point we had a functional caching FTP server, which we put into service.
6. A Web-based interface to the virtual tree. This turned out to be much more work than expected, mainly because of the one-shot nature of Web client requests which conflicts with our need to hold on to expensive resources like open connections to FTP servers.
7. A Web proxy interface to the virtual tree, which allowed people to configure their browsers to use CFTP automatically for all FTP URLs.

4 Analysis of New System

4.1 Deployment

The first application that was ready for deployment was the FTP server (stage 5 in the Implementation History above), but by this time this was ready it was clear that the restrictions imposed by the FTP protocol would make the system unnecessarily unfriendly to users. In particular it offered no easy way of checking whether a directory listing was stale, or of refreshing a listing. Experience with the need for the `reload` button in Web browsers convinced us that this would be a real problem for users.

The initial announcement of the availability of the Web interface to the UK academic community generated an initial burst of interest but, despite positive reaction from individual users to the interface, usage levels were disappointing.

We then used the Web interface for access to the HENSA Unix FTP archive. HENSA Unix has a set of introductory web pages, with links off to various parts of the archive. The majority of these links were FTP URLs which, when selected, resulted in the standard browser FTP page display. We changed these links to use the caching FTP Web interface. For this use no caching was actually done – we just arranged in the configuration file to export local filestore on the FTP server.

We found that usage of the interface increased dramatically after this change. As we had suspected, users liked the interface when it was convenient to get to, but were not prepared to visit the server just to access a file via FTP. The conclusion was that we had to make it easier for users to access the service.

These considerations were the motivation for developing the proxy server interface, which makes it much simpler to use the caching FTP server. Following a one-time configuration of the browser, all FTP URLs are handled using the server.

The main problem that we now faced was that most users do not configure their own browsers: site administrators set up default arrangements for using proxy servers. Usually a site cache is used, which in turn may link to a national cache. The key to large-scale use of the service appeared to be to arrange for these caches to use our proxy for FTP requests. We thus arranged for the Squid site cache at the University of Kent to use Cftp for FTP URLs.

Eventually we would like to see the CFTP proxy server linked up to the national Web caches, although we will first need to gain some experience with how the system behaves under heavy load. It may be necessary to distribute the service over several hosts.

4.2 Usage Statistics

The initial usage of the service was disappointing, we think due to the lack of a convenient way to use the service. Links to Cftp were put on the HENSA Unix front Web page at the beginning of June and, at the end of September, the University

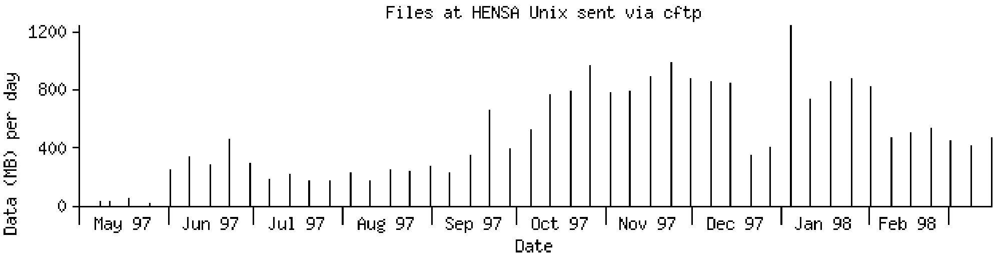

Figure 1: Files at HENSA Unix sent via Cftp

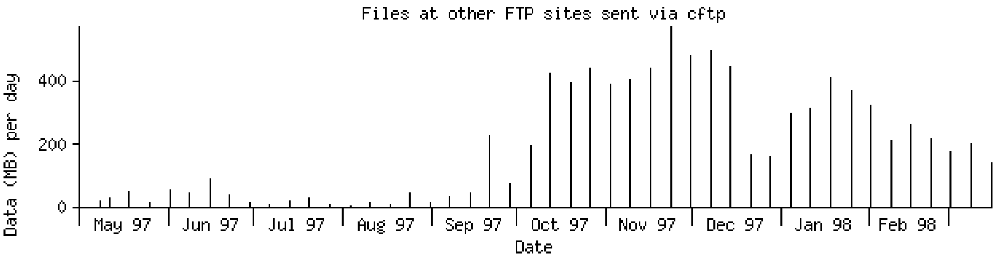

Figure 2: Files at other FTP sites sent via Cftp

of Kent's site Web cache was connected. This unfortunately had to be disabled in the new year as the version of Squid running on the cache was upgraded and new version was found to be incompatible with Cftp.

Figures 1 and 2 show the amount of data being shipped per week by Cftp acting as an interface to HENSA and as an ftp cache. The latter requests are those that are saving bandwidth, since they would otherwise have gone to the origin FTP server. The data volume levels for non-HENSA sites are in line with the requests per day, but those for HENSA files show more variation. The reasons for this are unclear, although the most likely factor, given the current usage levels, is that a few users pulling large files are making a substantial difference to the week's totals.

4.3 Hit rates

Table 9 shows a breakdown of requests by the way they were handled. The methods are

direct User explicitly requested HENSA Unix.

ftp Cache miss: user asked for a non-HENSA item, and we had to visit the origin server to fetch it.

cache Cache hit: as above, but we used a previously cached copy of the item.

mirror Another form of cache hit: user asked for a non-HENSA item, but we found an up-to-date copy of the item mirrored at HENSA Unix.

Roughly 62% of the usage so far has been for access HENSA Unix, but the proportion of non-HENSA accesses rose with the introduction of the link to the University of Kent's site cache. This trend has reversed since it was disconnected. Accesses to HENSA Unix via Cftp do not save international bandwidth, as they are merely accessing an existing resource via a new interface. Thus, we now consider

Method	Requests		Data(MB)	
direct	621544	62%	95514	61%
ftp	198630	20%	38999	25%
cache	170557	17%	19364	12%
mirror	10830	1%	3484	2%
All	1001561	100%	157363	100%

Table 9: Cftp accesses by method: May 8th 1997 - Mar 1 1998

Method	Requests		Data(MB)	
ftp	198630	52%	38999	63%
cache	170557	45%	19364	31%
mirror	10830	3%	3484	6%
All	380017	100%	61849	100%

Table 10: Non-HENSA accesses by method: May 8th 1997 - Mar 1 1998

the breakdown of requests excluding HENSA Unix. Table 10 shows the same data as Table 9 but excludes HENSA accesses.

As the tables show, the hit rate is running at about 46% by request, and 36% by volume of data. It is not clear why these rates are different, although it is presumably due to differences in access patterns for files and directories (the request counts include directory listings, whereas the data volume only counts file downloads).

The directory/file split shows up in the mirror statistics. Cftp will only use mirrored items to satisfy file requests; directory requests go via the cache or direct to the origin server. Mirrored files account for a quarter of the cache hits by volume, but there is plainly scope for more work in this area. We should for example take advantage of UK mirror sites other than just HENSA Unix.

Bandwidth savings are currently running at a few hundred megabytes per week and this represents savings of less than 1% of the total FTP traffic. This is obviously not enough to have a significant effect on international bandwidth, but with the system in place, we will now shift our efforts towards increasing usage levels for the future.

5 Future Extensions

The system is currently installed and running at HENSA Unix, but there is still much scope for improving it.

5.1 Extended support for mirroring

Currently CFTP knows about FTP resources mirrored at HENSA Unix, but there are many other mirror sites in the UK. Users must currently explicitly choose to use these sites rather than going direct to the origin servers. As our data analysis showed, many users fail to use local mirror sites.

There is obvious scope for gathering mirroring information in machine readable form so that CFTP and other systems can use it to make mirroring more transparent. Work in this area has started with the UK project to catalogue mirror sites [2]. At HENSA Unix we intend to use this information to integrate more mirror sites with CFTP.

This is an area where the site maintainers could help. Many FTP sites have files giving users information about their nearest mirror site. It would not take much to maintain a version of this in a format that a program could parse. With this in place, CFTP could automatically locate mirror sites without being configured in advance.

There is also scope for closer integration of mirroring and caching at HENSA Unix. The mirror software [8] gathers directory listings to decide what needs updating, but at present these are discarded after use. It would be useful to save these listings so that CFTP could use them, thus avoiding the need to contact the origin server.

5.2 Improved Web Interface

There is scope for improvement of the CFTP Web interface. As it appears that the overall look of a Web page is so important to users, we are considering a number of cosmetic changes to the interface. In addition, there is scope for improvements to its functionality especially in the interface to composite files like *tar* files. Currently, only *tar* files are decoded; other formats, in particular *zip* files, should also be handled. In addition, displaying the contents of these files as a directory tree rather than a list would be a worthwhile improvement.

The interface for downloading multiple files is currently rather rudimentary, and could certainly be improved. As it stands now, the user selects a group of files then clicks the `Download` button, whereupon he/she is presented with a *tar* file containing the selected files. It would be better to present the selected list as a fresh Web page, with buttons offering a selection of file formats (*tar*, *zip*, compressed or not). The page could also be used to refine the selection of which files to download.

We are also considering the production of a squid-like interface in order to make the system more consistent with the National Web Cache.

Finally the use of Java [5] should be investigated. This has the potential to make the Web interface much more interactive than vanilla CGI allows. It also offers the possibility of retaining per-user state in a much more convenient way. But there are pitfalls as well; not all users (or sites) are prepared to enable Java in their browsers, and the security restrictions in Java may get in the way of providing a capable interface.

5.3 Support for ICP

Currently there is no support in CFTP for ICP [11], as used by the squid cache. CFTP still interoperates with squid, because ICP support is not mandatory, but in the current form squid caches cannot detect failure of the CFTP cache. All that Squid can check is that the CFTP server host is responding to pings, so if the host is running but the CFTP server is not, the user can see long delays and timeouts.

ICP support should not be hard to add, as the protocol is quite simple.

5.4 Distributed caches

The current CFTP system runs on a single host. While this copes with the current load, it obviously has scaling problems. There is nothing in the current system to prevent running it on several hosts, probably with a DNS based load sharing scheme, although this would lead to the cache contents being duplicated. It would, thus, be useful to add co-operative caching, whereby each server would have some knowledge about what files are in its peer servers' caches.

A currently used alternative to DNS based load sharing is for each cache to handle part of the URL space. There is support in the Web browsers for this, in that a browser can download a script which maps URLs to proxy servers. The problem with this is that the partitioning must be done manually by the proxy server administrators, and it is very difficult to set up and maintain a balanced load sharing scheme.

These considerations make the idea of automatic load balancing attractive. This would work at the server end by some form of negotiation over the ownership parts of the URL space. There would need to be a private protocol between peer servers to support messages like "I am now handling all URLs below `ftp.foo.com` – forward any such requests to me". Designing such a protocol would obviously be significant work. We would also need to implement load measuring on each server, so that each server could determine whether it should take on or shed load.

6 Conclusions

An analysis of the log data obtained from a number of sources has shown that there is scope for improving the effectiveness of the caching and mirroring of FTP servers. CFTP, the caching FTP server, described in this paper is a step towards this in that it may be configured to have a knowledge of local (national) mirrors as well as its own cache of FTP objects. This means that users benefit from local mirrors even though they may be unaware that the data they are requesting is available locally.

Initial trials of the software appear to support the log analysis and users have been positive in their reactions both to the caching software and the new Web interface to the FTP file tree.

The software is now installed as the Web interface to the HENSA Unix FTP archive and has been successfully connected to the University of Kent's local Web cache. A number of improvements to the software and its interface have been identified.

We believe that the server is capable of coping with the load generated by the National Web Cache and that it can offer a significant saving in both international bandwidth and the time taken to retrieve data.

7 Acknowledgements

This work was funded under the UKERNA project "Efficient Use of International Bandwidth for Information Access", managed by Dr. John Kwan.

We would like to acknowledge the help given by the busy FTP site administrators who agreed to run our log analysis scripts. We are particularly grateful to David C Lawrence (at ftp.uu.net) and Nash Foster (at sunsite.unc.edu) for helping us with raw data from their logs over many months.

References

[1] M.L. Bowman, T.R. Hopkins, and N.G. Smith. HENSA Unix: serving the UK. *Axis*, 2(2):6–11, 1995.

[2] Manchester Computing. Registry of FTP mirror sites on JANET. http://www.ftpregistry.mcc.ac.uk/.

[3] Netscape Communications Corporation. Netscape proxy server. http://home.netscape.com/comprod/server_central/product/proxy/.

[4] R. Fielding, U. C. Irvine, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol — HTTP/1.1. Technical Report RFC 2068, http://www.hensa.ac.uk/ftp/mirrors/uunet/.vol/2/inet/rfc/rfc2068.Z, January 1997.

[5] M. Grand. *JAVA Language Reference*. O'Reilly, Sebastapol, California, 1997.

[6] IEEE/ANSI. *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language]*, Std 1003.1 edition, 1996.

[7] B. Kantor and P. Lapsley. A proposed standard for the stream-based transmission of news. Technical Report RFC 977, http://www.hensa.ac.uk/ftp/mirrors/uunet/.vol/2/inet/rfc/rfc977.Z, February 1986.

[8] L. McLoughlin. mirror.pl. ftp://src.doc.ic.ac.uk/computing/archiving/mirror/mirror.tar.gz.

[9] J. Postel and J. Reynolds. File transfer protocol. Technical Report RFC 959, http://www.hensa.ac.uk/ftp/mirrors/uunet/.vol/2/inet/rfc/rfc959.Z, October 1985.

[10] D Wessels. Squid internet object cache. http://squid.nlanr.net/Squid/, 1998.

[11] D. Wessels and K. Claffy. Internet cache protocol (ICP), version 2. Technical Report RFC 2186, http://www.hensa.ac.uk/ftp/mirrors/uunet/.vol/2/inet/rfc/rfc2186.Z, September 1997.

Vitae

Mark Russell graduated in Computer Science at the University of Kent in 1985. He worked on the Kent Software Tools project, developing a file browser, the ups debugger and a portable windowing library. He joined the Systems Group at Kent in 1989, where he spent his time defending Unix systems against hordes of students.

From 1995 to 1997 Mark worked on the UKERNA project "Efficient Use of International Bandwidth for Information Access". He now supports the HENSA service at the University of Kent.

Tim Hopkins is a Reader in Numerical Computing at the University of Kent, UK. He obtained a Ph. D. from the University of Liverpool in 1977 and has been Algorithms Editor for ACM Transactions on Mathematical Software since 1994.

He has managed the UK National Academic Unix Software Archive since 1989 and was involved in the setting up of the original UK National Academic Web Cache.

His research interests also include software quality and testing, parallel numerical computing and simulations.