# Usenetfs: A Stackable File System for Large Article Directories

Erez Zadok and Ion Badulescu
*Computer Science Department, Columbia University*
{ezk,ion}@cs.columbia.edu

**CUCS-022-98**

## Abstract

The Internet has grown much in popularity in the past few years. Numerous users read USENET newsgroups daily for entertainment, work, study, and more. USENET News servers have seen a gradual increase in the traffic exchanged between them, to a point where the hardware and software supporting the servers is no longer capable of meeting demand, at which point the servers begin "dropping" articles they could not process. The rate of this increase has been faster than software or hardware improvements were able to keep up, resulting in much time and effort spent by administrators upgrading their news systems.

One of the primary reasons for the slowness of news servers has been the need to process many articles in very large flat directories representing newsgroups such as *control.cancel* and *misc.jobs.offered*. A large portion of the resources is spent on processing articles in these few newsgroups. Most Unix directories are organized as a linear unsorted sequence of entries. Large newsgroups can have hundreds of thousands of articles in one directory, resulting in significant delays processing any single article.

Usenetfs is a file system that rearranges the directory structure from being flat to one with small directories containing fewer articles. By breaking the structure into smaller directories, it improves the performance of looking for, creating, or deleting files, since these operations occur on smaller directories. Usenetfs takes advantage of article numbers; knowing that file names representing articles are composed of digits helps to bound the size of the smaller directories. Usenetfs improves overall performance by at least 22% for average news servers; common news server operations such as looking up, adding, and deleting articles are sped up by as much as several orders of magnitude.

Usenetfs was designed and implemented as a stackable Vnode layer loadable kernel module[Heidemann94, Rosenthal92, Skinner93]. It operates by "encapsulating" a client file system with a layer of directory management. To the process performing directory operations through a mounted Usenetfs, all directories appear flat; but when inspecting the underlying storage that it manages, small directories are visible.

Usenetfs is small and is transparent to the user. It requires no change to News software, to other file systems, or to the rest of the operating system. Usenetfs is more portable than other native kernel-based file systems because it interacts with the Vnode interface which is similar on many different platforms.

## 1 Introduction

USENET is a popular world-wide network consisting of thousands of discussion and informational "news groups." Many of these are very popular and receive thousands of articles each day. In addition, many control messages are exchanged between news servers around the world, a large portion of which are article cancellation messages generated by anti-spam detection software. All of these articles and messages must be processed fast. If they are not, new incoming articles may be dropped.

Traditional Unix file system directories are structured as a flat, linear sequence of entries representing files. When the operating system wants to lookup an entry in a directory with $N$ entries, it may have to search all $N$ entries to find the file in question. Portions of directories are often cached in the file system, so that subsequent lookups do not have to retrieve the data from disk. Table 1 shows the frequency of all file system operations that use a pathname on our news spool over a period of 24 hours.[1] The table shows that the bulk of all operations are for looking up files, so these should run very fast regardless of the directory size. Operations such as creating news files and deleting ones are usually run synchronously and account for about 10% of news spool activity; these operations should also perform well on large newsgroups

These requirements necessitate a powerful news server that can copy memory fast, and have fast disks and I/O. As demands grow, the ability of the news server to process arti-

[1] Table 1 does not include reading and writing operations since they do not use a pathname in the file system function. In the file system, a lookup operation precedes every read of an existing file.

| Operation | Frequency | % Total |
|---|---|---|
| Readdir | 38371 | 0.48 |
| Lookup | 7068838 | 88.41 |
| Unlink | 432269 | 5.41 |
| Create | 345647 | 4.32 |
| All other | 110473 | 1.38 |
| **Total** | 7995598 | 100.00 |

Table 1: Frequency of File System Operations on a News Spool

cles diminishes to a point where it starts rejecting or "dropping" articles. The effort to upgrade a site's news server is significant: large amounts of data need to be copied to a new server as fast as possible, because while an upgrade is in progress, new articles are not processed and can be lost.

In practice, many sites have resorted to reducing the number of articles in use by removing large newsgroups from their distribution and expiring articles more often, sometimes as often as several times a day. Most site administrators accepted the fact that their news servers will lose articles on occasion.

For example, our department runs an average size news server. We have several hundred users and three feeds from neighboring sites. Our server has had two major upgrades in the past 5 years, and several smaller upgrades in between. The major upgrades were from SunOS 4.1.3, to Solaris 2.x, and the last one was to Linux 2.0. Each major upgrade included news server (INN) software upgrade, a faster CPU, more memory, and more and faster disk space. Our previous news server was running on a Sun SparcStation 5 with 8GB of stripped disk space, 196MB of RAM, and Fast Ethernet. But the CPU and I/O bus had not been able to keep up with traffic, and for the last two years of that server's life, it kept on losing more and more articles. Just before it was replaced, our old news server was dropping 50% of all articles.

A few months ago we upgraded our news server to an AMD K6/200Mhz with faster disks and tripled the overall disk space available. We used the top-of-the-line SCSI cards and Fast Ethernet adapters. We also upgraded the operating system to Linux 2.0.34, because the Linux operating system is a small, fast, and highly optimized for the x86 platform. In addition, Linux's disk based file system (ext2fs) has two features useful for optimizing disk performance:

1. It can turn off the updating of access times of files in the inodes; access times are not useful for news systems.

2. While ext2fs' on-disk directory structure is linear, it hashes cached entries in kernel memory for faster access.

Since the upgrade, our new news server had dropped no articles, and has kept up with traffic. However, we have noticed that its network utilization is over 80% and that more disk space is constantly being added. At the current growth rate, we expect it to outrun its capabilities in a couple of years.

## 1.1 Current Solutions

Several current solutions are available to the problem of slow performance of large directories used with news servers. They fall into one of two categories:

1. Modified news servers that store articles in an alternate fashion[Fritchie97].

2. New native file systems that arrange directory entries in a manner that is accessible faster than linear search time[Reiser98, Sweeney96].

These solutions suffer from several problems.

- **Development costs and stability**: news server software is large and complex. It is difficult and time consuming to modify it. Creating new native disk-based file systems is even more complicated, requiring deep understanding of the operating system internals. Such software will take time to become stable, and administrators would be reluctant to use it initially.

- **Portability**: INN is a user-level software, and needs to be portable to many platforms. Fundamental changes to it such as a new storage methodology are not simple and require porting to existing platforms. Worse, native file systems are not portable at all, since they directly interface with operating system internals, device drivers, and the virtual memory subsystem. A solution that is portable to only a few platforms will not enjoy wide usage.

- **Deployment**: distributing a rather different news server software or file system is a large undertaking. Since the changes are so fundamental, an upgrade path may not be a drop-in replacement for neither the news server nor the file system. News administrators would be reluctant to make significant changes to their news server unless the benefits would be significant.

Our approach modifies neither the news server/client software nor the native file systems.

## 1.2 The Stackable Vnode Interface

Usenetfs is a small file system based on the loopback (lofs)[SMCC92] one. Usenetfs mounts ("stacks") itself on top of a news spool hierarchy and interfaces between existing news software and disk based file systems, a seen in Figure 1. It makes a hierarchy of many small directories appear to be a single large flat directory.

"Vnode Stacking"[Heidemann94, Rosenthal92, Skinner93] is a technique for modularizing file system functions, by allowing one vnode interface to call another. Before stacking existed, there was only a single vnode interface; higher level operating system code called the vnode interface which in turn called code for a specific file system. With vnode stacking, several vnode interfaces may exist and may call each other in order. The Usenetfs and vnode layers in Figure 1 are really at

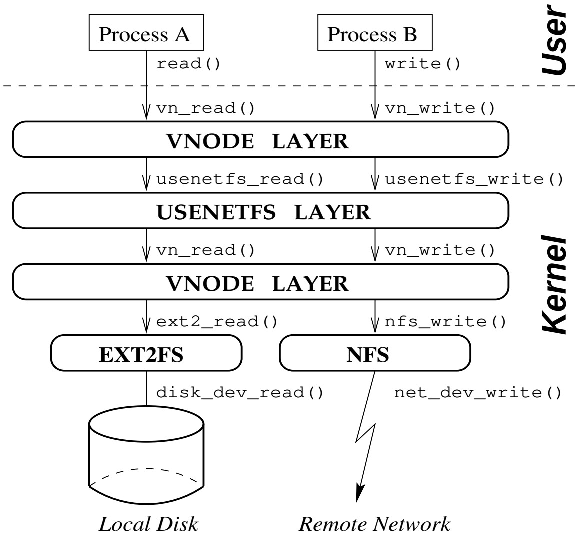


Figure 1: Stacked Vnode File System

the same abstraction level; each one may call the other interchangeably.

## 2 Design

Our design goals for Usenetfs were:

- Usenetfs should not require changing existing news servers, operating systems, or file systems. It should maintain a valid file system on the one being managed. At the same time, Usenetfs should be as portable as possible.
- It should improve performance of these large directories enough to justify its overhead and complexity.
- It should be small and impose little overhead.
- It should allow selective management of large directories without requiring that smaller ones be managed as well. This was to allow finer grained control over which newsgroups in the news spool are managed by Usenetfs and which are not.

The main idea for improving performance for large flat directories was to break them into smaller ones. Since article names are composed of sequential numbers, we took

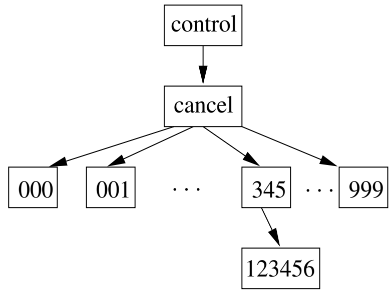


Figure 2: A Usenetfs Managed Newsgroup

advantage of that. We decided to create a hierarchy consisting of one thousand directories as depicted in Figure 2. We therefore distribute articles across 1000 directories named 000 through 999. Since article numbers are sequential, we maximize the distribution by computing the final directory into which the article will go based on three lesser significant digits, skipping the least significant one. For example, the article named `control/cancel/123456` is placed into the directory `control/cancel/345/`. The article name itself does not change; it only gets moved one level down. We chose to pick the directory based on the second, third, and fourth digits of the article number to allow for some amount of "clustering." By not using the least significant digit we cluster 10 sequential articles together; e.g. the 10 articles 123450-123459 get placed in the same directory, which increases the chances of kernel cache hits due to the likelihood of sequential access of these articles — a further performance improvement. In general, every article numbered X..XYYYZ gets placed in a directory named YYY.

Each operation that needs to manipulate a file name (such as `lookup`) performs name translation of the pathname to include the new subdirectory. For reading a whole directory (`readdir`), we would iterate over all of the subdirectories in order: 000, 001, ..., 999. Each entry read in these directories is returned to the caller of the system call.

Usenetfs needs to determine if a directory is managed or not. We decided to use a seldom used mode bit for directories, the setuid bit, to flag a directory as managed by Usenetfs. Using this bit allows the news administrator to control which directory is managed by Usenetfs and which is not, using a simple `chmod` command.

The next bit of design needed was how to handle files and directories whose names are not article numbers. Directories containing articles may include other directories representing other newsgroups, threads database files, etc. Usenetfs optimizes performance for the majority of files in the news spool — articles. We decided not to complicate the code for the sake of these few non-article files: all non-articles are also moved one directory level downward into a special directory named `aaa`. For example, an original file `control/cancel/foo` is moved to `control/cancel/aaa/foo`.

We chose to move all files, articles, and non-articles, one level deeper because it was simpler to uniformly treat all files in a directory managed by Usenetfs. It simplifies lookups of the ".." parent directory. In order to maintain the illusion of a flat directory, Usenetfs also has to skip upward lookups of ".." one level up. For example, if a lookup for ".." happens in a managed subdirectory `control/cancel/345/`, that lookup must return the directory vnode for `control/` and not for `control/cancel/`. In other words, the process performing a lookup (or chdir) for ".." for a managed directory `control/cancel/` should not know that the underlying storage was different and should get back the expected directory `control/`.

An alternate solution to this problem that we considered was to manage article files based on their name, consisting only of digits, and assume that anything that does not consist of digits alone is not an article. However, there are unfortunately some newsgroups where part of the component name is all digits: *alt.2600* and *alt.autos.porsche.944* among others. Distinguishing between directories with numeric names and files with numeric names would require a stat(2) of each, and that would have slowed the performance of Usenetfs. Therefore we rejected this idea and opted for simplicity: move all articles and non-articles to a two-level deeper hierarchy.

The next issue was how to convert an unmanaged directory to be managed by Usenetfs — creating some of the 000-999 subdirectories, and moving existing articles to their designated location. When measuring the number of articles in various newsgroups over a period of one month, we noticed that the large newsgroups remained large, while the small remained relatively small; no major changes were noticed other than a small but gradual increase in traffic. Newsgroups that were good candidates for management by Usenetfs were not likely to become low traffic overnight and will continue to have lots of traffic. Also, the number of such large newsgroups is small. In our news server for example, only 6 out of 11,017 newsgroups contained more than 10,000 articles each. Given that, we decided to make the process of turning directory management on/off an off-line process to be triggered by the news administrator with a script that we provide.

Alternately, we could have put all that code in the kernel, but that would have complicated the file system a lot, and would have cost us in significantly more development time, since kernel work is difficult. We did not feel that it was crucial to include this functionality at this stage, especially since we did not expect many directories to be managed.

# 3 Implementation

The implementation of Usenetfs proceeded according to the design. We began by using a loopback file system (lofs) and modified it to our needs.

Each Vnode operation that handles a file name such as `lookup`, `open` and `unlink` calls a simple function that converts the file name to a one-level deep directory hierarchy. For example here is the (slightly simplified) vnode operation to remove a file:

```
usenetfs_unlink(inode_t *dir, char *name, int len)
{
  int err = -EPERM;
  inode_t *i_dir = get_interposed_vp(dir);

  if (dir->i_mode & S_ISUID) {
    err = get_transit_dir(name, len, &i_dir);
    if (err < 0)
      return err;
    err = i_dir->i_op->unlink(i_dir, name, len);
  }
 return err;
}
```

The function retrieves the stacked on (interposed) vnode from the current directory vnode. It continues by checking if the directory where the file `name` needs to be removed is managed by Usenetfs (whether the setuid bit is on.) If so, it calls the routine `get_transit_dir` to get the vnode for the directory where the file is actually located. This routine looks at the file name and computes the directory where the file should be. For example if we want to remove the file name `987`, this function gets the vnode for the directory `0987`; that value is put into `i_dir`. Finally, the unlink function calls the same operation on the interposed directory and returns its result.

## 3.1 Directory Reading

The one complication we were faced was with the "readdir" vnode operation. Readdir is implemented in the kernel as a restartable function. A user process calls the `readdir` C library call, which is translated to repeated calls to the `getdents(2)` system call, passing it a buffer of a given size. The buffer is filled by the kernel with enough bytes representing files within a directory being read, but no more. If the kernel has more bytes to offer the process (i.e. the directory has not been completely read) it will set a special EOF flag to false. As long as the user process sees that the flag is false, it must call `getdents(2)` again. Each time it does so, it will read more bytes starting at the file offset of the opened directory as was left off from the last read.

The important issue with respect to directory reading is not how to handle the file names, but how to continue reading the directory from exactly the offset it was left off the last time. Since the readdir kernel function needs to be implemented as a restartable call, the file system has to store some state in one of the returning variables or structures so that it may be passed back to the readdir call upon the next invocation; at that time the call must continue reading the directory exactly where it left off previously.

We chose Linux as our first development platform for one main reason: directory reading is simpler in the Linux kernel. In other operating systems such as Solaris, we have to read a number of bytes from the interposed file system, and parse them into chunks of `sizeof(struct dirent)` that have the actual file name characters appended to. It is cumbersome and asks the file system to perform a lot of bookkeeping. In Linux, much of that complexity was moved elsewhere to more generic code that is outside the main implementation of the file system. A file system developer would provide the Linux kernel a callback function for iterating over the entries in a directory. This function will be called by higher level code on each file name. It was easier for us to provide such a function, which in conjunction with our version of readdir proceeded to read directories as follows:

1. Generate an entry for "." and ".." and return those first.
2. Read the special directory "aaa" and return entries within in the order they were read.
3. Read all the directories 000, 001, through 999 and return entries within, also in order.

# 4 Evaluation

Our evaluation concentrated on three aspects: stability, portability, and performance. Of those, performance is the most important and is evaluated in detail.

## 4.1 Stability

We configured a test new server running INN version 1.7.2 and gave it a full feed from our primary news server. We turned on Usenetfs management for 6 large newsgroups and let it run. The system ran for two weeks without a single crash or indication of abnormal behavior. We then repeated the test on our production news server and got the same results. We were satisfied that the Usenetfs file system kernel module was stable.

## 4.2 Portability

When we started working with vnode-stackable file systems, we first searched for sources to a loopback filesystem; lofs is a good starting point for any stacking work. Linux does not have an lofs as part of the main kernel. We were able to locate a reference implementation of it elsewhere, but had to spend some time getting familiar with it and fixing bugs.[2]

[2]This time was actually spent while writing another more complex stackable file system, Cryptfs, which encrypts files transparently.

Being familiar with kernel and file system internals, and knowing how difficult and time consuming operating system work can be, we expected the implementation to take us several weeks. However, we completed the code for Usenetfs in one day. The speed at which this was accomplished surprised us. The two contributing factors to this were the simple design we chose and Linux's easier-to-use vnode interface (especially for the readdir function.)

Given sources to an lofs for another Unix operating system, we expect to be able to port Usenetfs to the new platform within a few days to two weeks. We know it will take longer than Linux because directory reading is more cumbersome in operating systems such as Solaris and BSD-4.4. We have ported other stackable file systems to Solaris and Linux; the most complex was an encrypting file system, and it was more complicated than Usenetfs. Nonetheless, our experience has shown that once an initial stackable file system is written, it can be ported in less than 2-3 weeks to another platform.

## 4.3 Performance

When measuring the performance of Usenetfs we were concerned with these three aspects:

1. What is the overhead of stacking alone?
2. How much does performance improve for the typical file system actions that a news server performs?
3. How much better does a news server run when using Usenetfs?

Usenetfs is a stackable file system and adds overhead to every file system operation, even for unmanaged directories. We wanted to make this overhead as small as possible. To test the overhead, we compared the time it took to compile Am-utils[3]. We ran a full configure and build 12

| File System | Linux 2.0.33 (32MB RAM) | |
|---|---|---|
| | SPARC/85Mhz | P5/90 |
| ext2fs | 1097.0 | 524.2 |
| lofs | 1110.1 | 530.6 |

Table 2: Time for 12 Large Builds (Sec)

times on an otherwise quiet system, and averaged the measured elapsed time for each build. The results are shown in Table 2. The overhead of the loopback file system, and therefore of a single level stackable file system, is 1.2%. This overhead is relatively small compared to the overall improvements that Usenetfs offers.

Next, we tested specific file system actions. We set a testbed consisting of a Pentium-II 333Mhz, with 64MB of ram, and a 4GB fast SCSI disk for the news spool. The

[3]Am-Utils is the new version of the Berkeley Automounter available from http://www.cs.columbia.edu/~ezk/am-utils/.

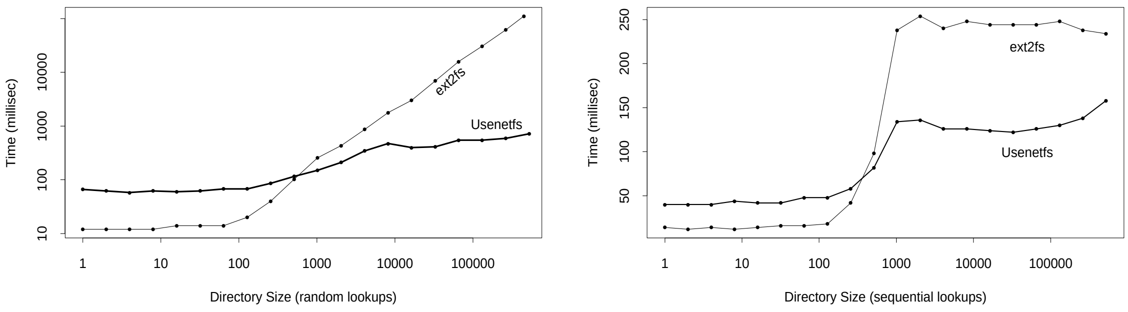


Figure 3: Cost for 1000 Article Lookups

machine ran Linux 2.0.34 with our Usenetfs. We created directories with exponentially increasing numbers of files in each: 1, 2, 4, 8, and so on. The largest directory had 524288 ($2^{19}$) files numbered starting with 1. Each file was exactly 2048 bytes long and was filled with random bytes we read from `/dev/urandom`. The file size was chosen as the representative most common article size on our production news server. We created two hierarchies with increasing numbers of article in different directories: one flat and one managed by Usenetfs.

The next tests we performed were designed to match the two actions most commonly undertaken by a news server. First, a news server looks up and reads various articles, mostly in response to users reading news, and when outgoing feeds are processed. The more users there are the more random the article numbers read would be, and while users read articles in a mostly sequential order, the use of threaded newsreaders results in more random reading. The (log-log) plot of Figure 3 shows the performance of 1000 random lookups (using `lstat(2)`) in both regular (unmanaged) and Usenetfs-managed directories, as well as the performance of 1000 sequential lookups. The time reported is in milliseconds spent by the process and the operating system on its behalf.

For random lookups on directories with fewer than 1000-2000 articles, Usenetfs adds overhead and slows performance. This was expected because the "bushier" directory structure Usenetfs maintains has over 1000 subdirectories. However, as directory sizes increase, lookups on flat directories become linearly more expensive while taking an almost constant time on Usenetfs-managed directories. The difference exceeds an order of magnitude for directories with 10,000 or more articles. For sequential lookups on managed directories with about 500 or less articles, Usenetfs adds a small overhead. When the directory size exceeds 1000, lookups on regular directories take twice as long. The reason the performance flattens out for sequential lookups is because cache hits are more likely due to locality of files in disk blocks. Usenetfs performs better because its directories contain fewer files so initial lookups cost less than on unmanaged directories.

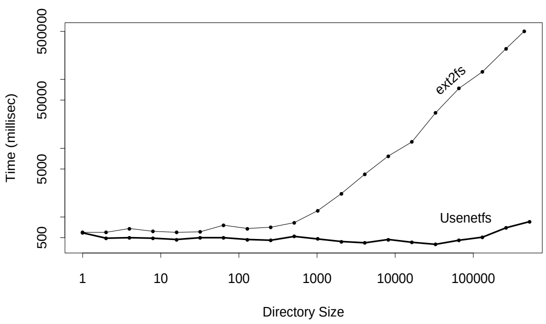


Figure 4: Cost for 1000 Article Additions and Deletions

The second action a news system performs often is creating new article files and deleting expired ones. New articles are created with monotonically increasing numbers. Expired articles are likely to have the smallest numbers so we made that assumption for the purpose of testing. Figure 4 (also log-log) shows the time it took to add 1000 new articles and then remove the 1000 oldest articles for successively increasing directory sizes. The results are more striking here: Usenetfs times are almost constant throughout. Adding and deleting files in flat directories, however, took linearly increasing times. Note that in both Figures 3 (random lookups) and 4, the linear behavior of the graph for the "regular" (ext2fs) file system is true when the number of articles in the directory exceeds about one hundred; that is because up until that point, all directory entries were served off of a single cached directory disk block.

Creating over 1000 additional directories adds overhead to file system operations that need to read whole directo-

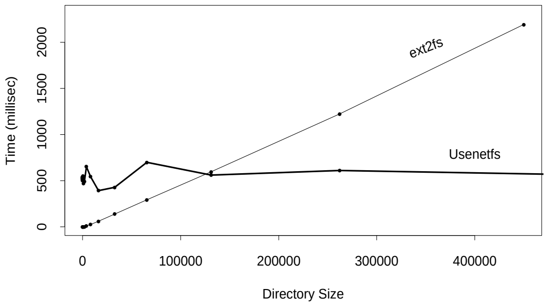


Figure 5: Cost for Reading a Directory

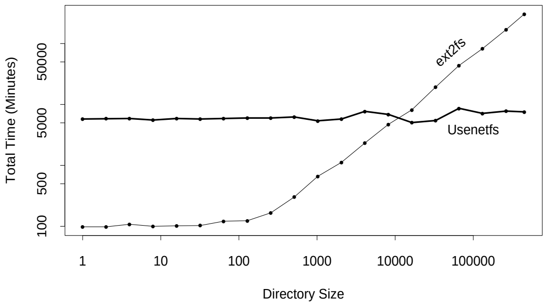


Figure 6: Potential Impact of Usenetfs on a News System

ries, especially the readdir call.[4] Figure 5 shows that readdir time is linear with the size of the flat directory. Usenetfs has an almost constant overhead of half a second, regardless of directory size; that is because Usenetfs always has at least 1000 directories to read. When directories contain more than about 100,000 articles, non-Usenetfs performance becomes worse. That is because such large directories require following double-indirect pointers to data blocks of the directory, while Usenetfs directories are small enough that no indirect blocks are required.

Figures 3 and 4 showed us that Usenetfs is beneficial for newsgroups with more than 1000 articles. Figure 5 showed us that readdir performance is better for directories with more than 100,000 articles. It is clear from these figures that such large directories are very well suited to be managed by Usenetfs, and at the same time that it is not worth managing directories smaller than 1000 articles. But when directory sizes are between 1000 and 100,000 we have to find out if the benefits of Usenetfs outweigh its overhead. Figure 6 shows the total time possibly spent by a news system over a period of 24 hours, taking into account Figures 3, 4, and 5 as well as the frequencies of file system operations reported in Table 1. The numbers were computed by multiplying frequencies by their respective times, and summing them for all operations, for each directory size. These computations assume that the number of operations reported in Table 1 were proportionally distributed among the newsgroups based on the newsgroup's size. Figure 6 shows that Usenetfs' benefits outweigh its overhead for newsgroups with 10,000 or more articles; at that size, the extra cost of readdir operations is offset by the savings in the other operations.

It should be noted here that the numbers reported in these figures assume that the directories of these various sizes are actually managed by Usenetfs. This analysis shows that newsgroups with only 10,000 articles or more should be managed. The "bad" numbers with overhead reported for example in Figure 5 are valid only if those small newsgroups are managed by Usenetfs. In practice they will not be, and their overhead will be 1.2% as reported in Table 2.

The last test we ran was intended to measure how well a host performs when it is dedicated to running a news system and employs Usenetfs. Our production news server is an AMD K6/200Mhz with 16GB of fast-SCSI disks and 128MB of memory. We decided to turn on management on every newsgroup with more than 10,000 articles in it. There were 6 such newsgroups totaling about 300,000 articles, the largest of which had over 120,000 articles. That was 25% of all the articles in the spool at the time.

The news system is a complex one, composed of many programs and scripts that run at different times, and depends on external factors such as news feeds and readers. We felt that a simple yet realistic measure of the overall performance of the system would be to test how much reserve capacity is left in the server. We decided to test the reserve capacity by running a repeated set of compilations of a large package (Am-utils), timing how long it took to complete each build.

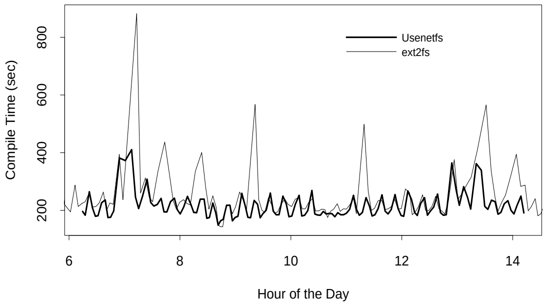


Figure 7: Compile Times on a News Server

Figure 7 shows the compile times of Am-utils, once when the news server was running without Usenetfs management, and then when Usenetfs managed the top 6 news-

[4] It also consumes at least 1000 more inodes per managed directory, but that is small compared to the rest of the news spool; the latter is often formatted with millions of inodes.

groups. The average compile time was reduced by 22% from 243 seconds to 200 seconds. The largest savings appeared during busy times when our server transferred outgoing articles to our upstream feeds, and especially during the 4 daily expiration periods. The largest expiration happens at 7am, when we also renumber the article ranges in the active file. Expiration of articles, and active file renumbering in particular, are affected more by the large newsgroups — the ones with the biggest directories and the most articles to remove. During these expiration peaks, performance improved by a factor of 2-3.

There are other peaks in the server's usage (around 7:45, 8:15, 9:30, and 11:20) representing bulk article transfers to our neighbors. These actions cause the lookup and reading of many articles from the spool, a lot of which reside in large newsgroups. When Usenetfs was used, these peaks were mostly smoothed out. The overall effect of Usenetfs had been to keep the performance of the news server more flat, removing those load surges.

We believe that the performance of Usenetfs can be made better. First, we have not optimized the code yet. Secondly, our server represents a medium size news site but the hardware is powerful enough that it can handle all of the traffic it gets. If we had more traffic, kept more articles online, or our news server had used a slower CPU, we believe that Usenetfs' improvements would be more pronounced. For example, one of our neighboring news sites (sol.ctr.columbia.edu) is a busier news server. It has 543 newsgroups with over 10,000 articles each, a few of which surpass 500,000 articles. These big newsgroups account for 87% of the spool's size on that server.[5] Such a busy server would benefit more from Usenetfs than our own.

Finally, we have experimented with other types of directory structures. For a given article numbered 123456, we tried to structure it using various subdirectories: 34/56/, 56/34/, and 456/. All three experiments resulted in poorer performance because they either had too many (10,000) additional directories thus imposing significantly larger overheads, or they did not cluster adjacent articles in the same directory and resulted in too many cache misses. All of the algorithms we have used are in the code base and can be turned on/off at run time. *These results will be included in the final paper.*

# 5 Related Work

## 5.1 Cyclic News File System

The Cyclic News File System (CNFS)[Fritchie97] stores articles in a few large files or in a raw block device, recycling their storage when reaching the end of the buffer. CNFS avoids most of the overhead of traditional FFS-like[McKusick84] file systems, because it reduces the need for many synchronous meta-data updates. CNFS reports an order of magnitude reduction in disk activity.

CNFS requires substantial modifications to INN 1.7, and is planned as part of INN 2.0. INN 2.0 in still under development and will take a long time to to become stable along with CNFS, and even longer to deploy throughout the Internet. Moreover, transition from the traditional storage method to CNFS will not be trivial. This underscores our point that any work that makes significant modifications to large news software packages will take a lot of time to become widely distributed.

## 5.2 XFS

XFS[Sweeney96] is Silicon Graphics' new file system for the IRIX operating system. It supports large file systems, a large number of files, large files, and very large directories. XFS uses B+ trees to store data and meta-data on disk, avoiding the traditional linear directory structure, and achieving good performance improvements over IRIX's older file system, EFS.

XFS comes only with IRIX and is not likely to be available for other operating systems. Since the requirements of news systems change often due to increasing resource demands, news administrators usually opt for less expensive off-the-shelf hardware that can be upgraded easily and inexpensively — mostly high-end PCs; they are not likely to purchase a more expensive SGI system.

## 5.3 reiserfs

Reiserfs[6] is a file system for Linux that uses balanced trees to optimize performance and space utilization for small and large files alike. By balancing files and their names, directories are packed more efficiently resulting in faster access.

Reiserfs is free but available only for Linux. It is a native disk-based file system and as such is very difficult to port to other operating systems. It is a complex and large piece of software that will take time to become stable and available for production use. (We have tried to test reiserfs but the kernel module crashed after creating some 50,000 files in one directory.)

# 6 Conclusions

For directories larger than 100,000 articles, Usenetfs improves performance by more than an order of magnitude. For directories containing 10,000 articles or more it improves performance by at least several factors. Usenetfs provides more efficient utilization of existing hardware, thus extending its lifetime in the face of ever increasing news traffic.

[5] Even if the number of large newsgroups is small, they often represent at least one quarter of all spool files — a significant portion — thus worth managing by Usenetfs.

[6] `http://www.idiom.com/~beverly/reiserfs.html`

We have shown that a relatively portable and highly stable file system can be written in a short period of time. The short development time and higher portability than native file systems were both the result of designing and implementing Usenetfs as a stackable vnode interface file system.

Stackable file systems and stackable vnodes are not a new idea, but they have seen very little use. With this work, we hope to prove that a lot can be accomplished with little effort and without the need to rewrite any application software, design radically new disk-based file systems, or redesign operating systems. We believe that many more stackable file systems can be written for existing vnode interfaces. But we also recognize that in the long run, a truly stackable vnode interface would have to be designed for modern operating systems in order to maximize its utility. We also believe that while more cumbersome and difficult to develop, new file systems coupled with a new data storage models for news are the better long term solutions to performance problems of USENET news systems.

Next we plan to complete our port of Usenetfs to Solaris and also port it to FreeBSD, since these are representative of the other two major Unix flavors in use.

## 7 Acknowledgments

Special thanks to Daniel Duchamp for his detailed reviews of this paper; his useful comments enabled us to improve Usenetfs' performance. Thanks to Alex Shender and Jeff Pavel for their assistance in the initial phases of the design, and to Alex for his comments on this paper. Also thanks to our ex-newsmasters Fred Korz and Seth Robertson for reviewing the work.

This work was partially made possible thanks to NSF infrastructure grants numbers CDA-90-24735 and CDA-96-25374.

## References

[Fritchie97] S. L. Fritchie. The Cyclic News Filesystem: Getting INN To Do More With Less. *System Administration (LISA XI) Conference* (San Diego, California), pages 99–111. USENIX, 26-31 October 1997.

[Heidemann94] J. S. Heidemann and G. J. Popek. File System Development with Stackable Layers. *Transactions on Computing Systems*, **12**(1):58–89. (New York, New York), ACM, February, 1994.

[McKusick84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[Reiser98] H. Reiser. Trees Are Fast. Unpublished Technical Report. The Naming System Venture, June 1998. Available http://www.idiom.com/~beverly/reiserfs.html.

[Rosenthal92] D. S. H. Rosenthal. Requirements for a "Stacking" Vnode/VFS Interface. Unix International document SD-01-02-N014. UNIX International, 1992.

[Skinner93] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. *USENIX Conference Proceedings* (Cincinnati, OH), pages 161–74. USENIX, Summer 1993.

[SMCC92] SMCC. lofs – loopback virtual file system. SunOS 5.5.1 Reference Manual, Section 7. Sun Microsystems, Incorporated, 20 March 1992.

[Sweeney96] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. *Usenix Conference Proceedings* (San Diego, California), pages 1–14. USENIX, 22-26 January 1996.

[Zadok97] E. Zadok. *FiST: A File System Component Compiler*. PhD thesis, published as Technical Report CUCS-033-97 (Ph.D. Thesis Proposal). Computer Science Department, Columbia University, 27 April 1997. Available http://www.cs.columbia.edu/~library/ or http://www.cs.columbia.edu/~ezk/research/.

## 8 Author Information

**Erez Zadok** is an PhD candidate in the Computer Science Department at Columbia University. His primary interests include operating systems, file systems, and ways to ease system administration tasks. The work described in this paper was first mentioned in his PhD thesis proposal[Zadok97]. In May 1991 Erez received his B.S. in Computer Science from Columbia's School of Engineering and Applied Science. In December of 1994 he received his M.S. degree in Computer Science from the same school. Erez came to the United States in 1987 and has lived in New York ever since. In his free time Erez is an amateur photographer, science fiction devotee, and classical music fan. Email address: `ezk@cs.columbia.edu`.

**Ion Badulescu** is a staff associate at the computer science department. He is also a B.A. candidate at Columbia University and is expected to graduate in May 1999. His primary interests include operating systems, networking, compilers, and languages. His life outside the Computer Science Department includes classical music, soccer, roller-blading, biking, and science fiction. Email address: `ion@cs.columbia.edu`.

For the latest news on Usenetfs and other stackable file systems, see http://www.cs.columbia.edu/~ezk/research/.