A File System Interface to the Internet*

Robert J. Minerick         Paul W. Schermerhorn
Vincent W. Freeh
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
{rmineric,pscherm1,vin}@nd.edu
January 17, 2001

Abstract

Nearly all computers today have access to Internet resources. However, the access of these resources typically requires specialized client programs. Therefore, users are forced to employ different tools when working with Internet resources than with local resources. This paper describes a system of user-level transformations for the Modify-on-access file system that map the local file system onto four different Internet resources. This allows all tools seamless access to the Internet through the file system interface. We also show that the standard Unix file system interface is sufficient to provide access to most of the resources on the Internet. Furthermore, this paper argues that accessing Internet resources through the file system offers advantages to both users and application programmers.

1 Introduction

Most applications do not have direct access to the wealth of resources available on the Internet. The Internet provides access to these remote resources through various access protocols. However, in order to access a remote resource, a single application must implement a distinct interface for each specific access protocol. For example, GNU Emacs [Sta94] provides access to many Internet resources as well as local files. It supports FTP and HTTP servers, net news (NNTP), email (IMAP, POP, etc.), and more. However, there exists a distinct Emacs module for each access protocol (such as, ange-ftp, w3, Gnus, RMAIL, VM, et al.). Although this provides Emacs access to Internet resources, no other application is able to leverage this work.

Consequently, an “Internet-aware” application must explicitly provide access to each type of Internet resource it chooses to support. Furthermore, only applications specifically written for the Internet can directly manipulate Internet resources. This paper presents a system that maps Internet resources onto the local file system. With a file system interface to the Internet, an ordinary (not “Internet-aware”) application can seamlessly access Internet resources.

Our system not only simplifies creating Internet-aware applications, it makes all existing applications and tools Internet-aware. A standard shell script can be a web crawler. A search engine can be constructed from find and grep. One can browse an Internet site with traditional Unix tools, such as ls and cd. One can view remote content with less or ghostview or parse a remote document with awk or sed. These examples are only a small sample of the potential uses of our approach.

The solution uses the Modify-on-access (Mona) file system [Ken98, KF99]. Mona is an active file system that operates on input and output data streams using modular transformations. Mona supports both

*This research was supported by an NSF CAREER grant (CCR-9876073), the Arthur J. Schmitt Fellowship, and the ND Faculty Research Program.
kernel-resident and user-level transformations. Transformations executing in the kernel, naturally, run faster, whereas user-level transformations are safer and easier to create. For this paper, we implemented four user-level transformations: FTP, HTTP, NNTP, and IMAP, each of which supports a different Internet access protocol. There are many more access protocols that could be supported similarly; however, these four constitute a significant amount of Internet traffic and a diverse set of applications.

Each Mona Internet transformation maps an Internet resource onto the local file system. This mapping is both a physical mapping that converts remote data into local data and a conceptual mapping that superimposes the file system metaphor onto an Internet resource. The former mapping is usually straightforward with the primary concerns being correctness and efficiency. The latter mapping, however, is difficult when the structure of the Internet resource differs from that of a file system. In such cases, there may be several reasonable choices, with no one mapping clearly considered to be the best.

The Mona file system and Internet transformations create a system that seamlessly accesses remote Internet resources. Consequently, standard (i.e., not Internet-aware) tools and applications work on Internet resources without modification. Additionally, application programmers can access the Internet through standard read and write system calls, without mastering the FTP, HTTP, NNTP, or IMAP protocols.

There are other systems that provide a file system interface to the Internet. However, each provides only one interface, supporting a single access protocol. Furthermore, most of these systems use a proxy server that sits between the client and the Internet resource. Our system is the only one we know of that supports multiple access protocols and resides entirely on the client.

The remainder of this paper is organized as follows. The next section presents an overview of the technologies involved and similar projects. Section 3 discusses four Mona transformations for accessing Internet resources. Section 4 presents the results of some tests of Mona and these transformations. Lastly, Section 5 discusses our conclusions and future directions.

2 Overview

This section presents important technologies related to our work. It presents other projects that provide a file system interface to the Internet. Next, it contains a brief overview of the Mona file system, which provides the basis for our Internet transformations.

2.1 File System Interfaces to the Internet

The Gecko NFS Web proxy provides access to HTML documents on HTTP servers [BH99]. The Gecko proxy sits between a client and a web server. Client requests to the virtual web file system on Gecko are translated into HTTP requests and sent to a web server. Through NFS, Gecko provides transparent access to any HTTP server. Gecko also provides increased efficiency compared to directly issuing HTTP requests from the client through caching. Gecko presents a very rational and complete conceptual mapping of HTTP to NFS. The Mona HTTP transformation utilizes a nearly identical mapping.

Similar to Gecko, the Alex file system is an NFS to FTP proxy [Cat92]. An Alex proxy server is positioned between a client and FTP servers. Alex provides transparent access to FTP servers and maintains cache consistency of files cached on the server.

Sun’s WebNFS supports accessing NFS file systems using HTTP [Ca96]. NFS is widely used in intranets, but essentially non-existent on the Internet. This is because NFS does not perform well on the Internet since the longer latencies experienced on the Internet cause inefficiencies. WebNFS addresses a different problem than the one we are trying to solve. It provides an HTTP interface to a file system and, consequently, the benefits of this approach are quite different.

WebFS has similar goals to those of Gecko [VEA96]. It is a loadable Solaris kernel module that provides a mapping of the URL name space into the file system name space. It uses a special client-side program to parse links and expand the name space. Although WebFS maintains compatibility with HTTP, the goal of WebFS is to replace it with a protocol that has support for cache coherence and authentication. Thus, even though it appears to be similar to this project (i.e., it is a client-side support program), it is quite different.
Microsoft Web Folders is a Web authoring component of Microsoft Internet Explorer. It enables users to manage files on a Web server using the Windows Explorer interface, enabling local manipulation of remote files. While this is similar to the problem we address, their solution applies only to HTTP and requires server-side support.

2.2 The Modify-on-Access File System

The Modify-on-Access file system is an active file system that performs operations on behalf of processes. In Mona, an application pushes and pops transformations, stream-oriented data operations, onto streams of data flowing to or from a process. When an application performs a read or write\(^\text{1}\) on a stream containing a transformation, the file system executes the transformation on all data flowing down the stream. As a result, Mona enables numerous file system extensions, such as compression or encryption.

The Mona file system supports in-kernel transformations that are pushed onto (or popped off of) files. In this sense, it is similar to STREAMS [Rit84]. However, while STREAMS is a character-oriented system, Mona is a block-oriented system and supports exporting computation outside of the kernel [KFSR00].

A transformation applied to a data stream creates a virtual file—a file whose data resides logically, but not physically, in the file system. A program creates transient virtual files by invoking an ioctl system call on an open file descriptor, for example:

```c
fd = open("base_file", O_RDWR);
fd = ioctl(fd, PUSH_INPUT, "decompress_xform");
```

In the above example, the base file is a compressed file. The virtual file, accessed through \(fd\), is an uncompressed version of the base file.

A persistent virtual file is created by a transformation link, which is similar to a symbolic link. In addition to the name of the base file, a transformation link contains names of transformations to be applied to the data streams. A transient virtual file exists until the file is closed, whereas a persistent virtual file (i.e., a transformation link) is stored in the file system.

Kernel-resident transformations add a modest amount of overhead (6-12\%) on a per-system-call basis. In all cases, the absolute overhead added is less than 5 \(\mu\)s. This overhead is the additional cost of executing a file system call guarded by the \(\text{noop}\) transformation, which does not perform any computation. The overhead

\(^{1}\)Mona also supports transformations on other file system operations, such as open.
experienced by an application is much less than these micro-performance numbers would suggest because applications do not consist only of file system calls. For example, the Andrew file system benchmark [HKM+88] slows down only 0.3% when every file is guarded by the \textit{nop} kernel transformation as compared to the native \texttt{ext2} file system. Similarly, PostMark [Kat97] and kernel compile experience overheads of 1.0% and 1.8%, respectively [KFSR00].

A user-level transformation executes in a user-space helper process. The kernel-resident \textit{export} transformation transfers blocks of data into user-space and receives modified blocks from a helper process. The Mona daemon process executes (in user-space) on open calls of files guarded by the export transformation. On open, the daemon forks a helper process, which will process data in the stream until the stream is closed. As expected, user-level transformations are slower than kernel-resident transformations. The cost of individual system calls using identical transformations is between 2 and 20 times greater for a user-level transformation than for a kernel-resident transformation [SKMF00].

Figure 1 shows how user-level transformations are executed. The data stream flows from left to right. Three kernel-resident transformations act on the stream. The middle transformation exports the blocks of data up to a user-level helper process, which processes input blocks from the kernel and returns resultant blocks to the kernel. The Mona daemon process creates a distinct helper process for each user-level transformation when a data stream is opened.

3 Details of Mona Internet Transformations

The mapping of Internet resources onto the local file system presents unique opportunities for both the user and the application developer. However, there are several issues that must be addressed to effectively access these resources through the file system interface. While certain issues are specific to individual protocols, there are three general issues that arise when mapping Internet resources onto the local file system.

First, a file system requests blocks of a small, fixed size from the disk. However, Internet resources are generally fetched all at once. Our solutions cache an entire remote resource (which is fetched in one request) locally on an \textit{open} system call. A subsequent \textit{read} system call copies fixed-size blocks from the local storage. While this can make opening large remote resources much slower than a local open, reads execute natively.

Second, the request could fail silently. Our solution here is to time out. Before issuing a remote request, each Mona transformation sets a timer. If the timer expires before a reply is received, the transformation either re-issues the request or returns an error code. The timer is set in the user-level helper process, not in the kernel.

Finally, the local cached data could become stale. Consequently, our solutions must address maintaining consistency. Cached resources are invalidated if they are too old. On an \textit{open} call, our solutions look for a local cached copy of the Internet resource. If none exists or it is stale, then a fresh copy of the resource is fetched. Client-side aging of files is not necessarily the best form of consistency. However, these Internet access protocols are request-reply; therefore, it is not possible for the server to asynchronously inform a client that a resource is stale. So client-side aging is perhaps the best solution one can provide for this environment. The proper age at which one should invalidate a cached resource will vary with the resource. For example, a page that presents the score of a baseball game should become stale in a few minutes, whereas a \texttt{README} file in a source distribution could remain fresh for days.

The next few sections present the four Mona Internet transformations and the issues involved in implementing them. We present the FTP transformation first because it is the most straightforward in both implementation and in concept. All of the transformations are implemented similarly and used identically, so only the FTP section discusses how to setup and use Mona transformations. The other sections only discuss features unique to the specific transformation.

3.1 The FTP Transformation

The File Transfer Protocol (FTP) provides access to a remote file system. It supports the reading, writing, and listing of remote files and directories. FTP requests are similar to file system requests. Of the four
protocols we discuss, the FTP protocol is most naturally mapped onto the local file system, both physically and conceptually.

Typically, users run an FTP client program to transfer files from and to a remote file system via an FTP server. This can be cumbersome. Suppose we want to preview a postscript file on an FTP server. After locating the desired file on the remote server, we transfer the file to the local file system using the appropriate client program command. Next, a local binary, such as *ghostview*, displays the file. Finally, we would usually remove the file because it is not needed, or it is in the wrong location.

Although accessing an FTP server with a special client program was cumbersome, it was adequate when access to remote files was infrequent. It is inadequate today when the network plays such a vital role in computing. The Mona FTP transformation provides seamless access to remote files. In the above example, the user simply executes *ghostview* on the virtual file, which is guarded by the FTP transformation. There is no need explicitly to transfer the file, nor does one have to clean up afterwards.

To use the FTP transformation, a user must initially create a transformation link that, when accessed, executes the FTP transformation. This is comparable to logging in with an FTP client program, except a Mona transformation link is persistent, whereas an FTP client will time out. It is through this transformation link that the remote FTP site will be accessed. A transformation link contains the name of the FTP transformation, the destination FTP site, the destination directory or file, and user name and password (if applicable).

To illustrate, we show the steps required. A symbolic link is simply a file that contains the name of another file in the system. A standard Unix file system understands that it must follow the link and open the named base file. In addition to the name of the base file, the Mona file system looks for any transformations that may be a part of the link name. The commands in the example below create a transformation link named `/ftp.ndlug.nd.edu` which “points” to a remote directory named `/pub` on the FTP server `ftp.ndlug.nd.edu`.

```bash
bash% touch .ndlug
bash% ln -s "./.ndlug " ftp_xform ftp.ndlug.nd.edu /pub \
-o ftp_xform ftp.ndlug.nd.edu /pub \
-x nop_xform \n-w fail_xform" ~/ftp.ndlug.nd.edu
```

First, an empty base file is created. Then, the transformation link is created with the standard `ln` command. The link “name” may contain extra fields specifying transformations. Newlines are used to separate fields in the base file name. Transformations are specified by flags in the target file name of the `ln` command, and there can be any number of transformations specified. In the example above, the `-o` argument instructs Mona to execute the transformation, `ftp_xform`, on the `open` system call. Mona passes the remaining arguments to the FTP transformation when it is executed. In the example above, the remaining arguments are the name of the remote FTP site and the directory or file on that FTP site. By default, `ftp_xform` logs in anonymously—additional arguments can specify a different user. The `-r` argument specifies the transformation to execute on the `read` system call. Because the base file is read unmodified, the read transformation is the `nop_xform`, which does nothing to the data. The `-w` argument specifies the transformation to execute on the `write` system call. Because we do not support write access to a remote FTP server, the `fail_xform` causes a write to fail.

When multiple transformations of the same type (i.e., read) are specified in a transformation link, Mona builds a transformation network that applies each transformation in turn in the order listed in the transformation link.

Once the initial setup is complete, the user may access the FTP site through the link as though it were part of the local file system. The link is persistent and will remain in the file system until explicitly deleted by the user.

---

2In example, the read transformation can be omitted. In fact, it should be because it is faster to access a file without any transformation than with `nop_xform`.

3This is not strictly needed since the FTP transformation would abort.
The FTP transformation is implemented as a transformation on the Linux open system call. When the link is accessed, for example by a user issuing the command cd "/ftp.ndlug.nd.edu", Mona first opens the transformation link and pushes the specified transformations on the appropriate streams. Then, because an open transformation is guarding this file, the transformation is executed. Because the remote file, /pub, is a directory, the ftpxform requests a directory listing from the FTP server. It parses the returned results and creates transformations links to all files contained in the directory listing. If instead /pub were an ordinary, non-directory file, then ftpxform would have fetched the contents of the file. In either case, the “contents” of a file (whether a directory listing or data) are fetched on open.

The ftpxform transformation uses two classes of files: transformation links and base files. Thus, a remote file (or directory) has two entries in the local file system. When ftpxform is executed, it first checks to see if the base file is empty. If the base file is empty, the remote file has not yet been fetched, in which case it transfers the remote file into the base file. If it is not empty, it then checks its age. If the cached copy is too old, then ftpxform fetches a fresh copy from the FTP server. The out-of-date time is user configurable. If the FTP file is a regular file, a base file is created in a .contents directory in the proper local directory. For a directory file, the base file is a directory with a name derived from the name of the remote directory.

The Mona FTP transformation seamlessly, but not transparently, provides access to FTP sites. It is seamless because any tool that accesses local files can access remote files. It is not transparent because one can see both the base files and the links in the local file system.

The primary use of FTP is browsing read-only file systems, although it does support writing. This transformation, however, does not support writing to remote FTP file servers. First, writing to an FTP site hardly ever occurs. Second, it is not clear when to issue remote writes. The local file systems writes incrementally, whereas FTP performs whole file transfers. It is quite inefficient to transfer a whole file to the FTP server for every local write of a file buffer. On the other hand, one could flush a modified file on close. But this changes the semantics of the file system and can lead to unexpected results. Although writing can be supported, its complexity is high and its utility low.

Several simple tools make it easier to use Mona for FTP transfers. The mountftp script creates the appropriate transformation link. Similarly, cleanupftp traverses the local FTP “mount” point (which contains cached copies of FTP files), discarding stale base files.

3.2 The HTTP Transformation

The Hypertext Transfer Protocol (HTTP) transfers hypertext documents that are written in Hypertext Markup Language (HTML). At its core, HTTP is a simple request-reply protocol. However, it is growing more complex as it evolves to handle new and unexpected demands—such as redirection, proxy servers, authentication, etc. Although supporting HTTP is straightforward, mapping a hypertext document into the file system metaphor is not.

The mapping of HTTP into the file system is more complex than the mapping of FTP. Like the other Mona Internet transformations, HTTP fetches entire files and can fail. However, there are two other issues that arise when mapping HTTP onto the file system.

Foremost is the the problem of file naming. Although a Web page name is similar in format to a Unix file name, it does not imply a hierarchical directory structure. While http://www.ndlug.edu/ppc/index.html may be a valid URL, http://www.ndlug.edu/ppc/ may not even be accessible. There are many issues involved in mapping a hypertext document into the file system. Baker and Hartman, [BH99], provide a discussion of these issues, as well as several possible solutions. They argue convincingly for a particular mapping, from which our implementation borrows heavily.

Unlike FTP, HTTP does not support listing a directory. Therefore, the mapping of URL’s onto the file system must be done slightly differently. Every Web page is represented by a Unix directory that reflects the structure of the document. The directory for each web page includes a .contents file reflecting the HTML contents, and a .headers file that contains the response headers sent from the server. If the contents of the document contains hyperlinks to other pages, a symbolic link is created for each of them.

The second issue involves the complex features of HTTP. When mapping HTTP into the file system
metaphor, some of the more complex features are lost. For example, header information plays an important role in determining exactly what is retrieved from a web server. It is common for an HTTP request to contain fields specifying the preferred language and browser. As Baker and Hartman point out, it is not practical to put options into the name of the file because options can be very lengthy, which makes them awkward to use and could exceed system limits on file length [BH99].

These lost features can be supported, but not within the standard file system name space. There are several ways to support these. For example, one could use auxiliary programs or the /proc interface to manipulate advanced features. Without loss of generality, we chose to use a differently configured Mona transformation for each set of features. Some features can be enabled with environment variables, but others are hard-coded into the implementation.

The HTTP protocol specifies a caching policy. Server-specified expiration allows the server to tell the client that the returned page is good until a specified time, at which point the client should invalidate the page and request a new one from the server. Our implementation uses this value to decide when to reload the page.

3.3 The NNTP and IMAP Transformations

The NNTP and IMAP transformations map nearly identically onto the file system, and share most of the same unique issues. Therefore, we discuss them together. The Network News Transfer Protocol (NNTP) provides access to news articles stored on a remote server. Newsgroups form a hierarchy of “dotted” names, which map clearly into the file system metaphor.

The Internet Message Access Protocol (IMAP) provides access to email located on remote mail servers. IMAP is popular, well featured, and supports remote mail reading. There are many mail protocols; however, with IMAP, one can access mail without migrating messages locally. This makes an email message a remote Internet resource. This protocol is simple, and the directory structure of a user’s mailbox maps cleanly into the file system.

Both these protocols use a simple request-reply model. An NNTP client can request a list of newsgroups, articles, or contents of articles, for example. The physical mapping of the both transformations is straightforward, and does not present unique issues beyond those of the other Mona Internet transformations.

The structure of the mapped NNTP and IMAP protocols is similar to that of an ordinary file system. NNTP groups and IMAP folders are both organized hierarchically, much like a file system. Our mapping is based on this hierarchy. As an example, suppose a user maps the NNTP server \texttt{mtnp.nd.edu} into the file system and retrieves the articles in the group \texttt{comp.sys.os.research}. To get to this group in the local file system, the user would change the current directory to \texttt{/mtnp/nd.edu/comp/sys/os/research}. When the user types \texttt{ls}, the \texttt{mtnp.xform} retrieves the article headers for the group and sets up the appropriate links in the directory. The user sees a listing of articles in the newsgroup. The names of the files are the articles numbers.

The IMAP transformation is nearly identical, except that its hierarchical structure is based on the folders and messages in a user’s remote mailbox. We cannot use the IMAP message numbers because IMAP servers renumber messages. Therefore, the transformation creates IMAP file names that are the time of receipt of the message. This naming scheme was chosen because the file name must be unique and it should refer to the same resource. Authentication is an issue with the IMAP transformation. The IMAP transformation can use either plain-text passwords stored in a private file, or it can authenticate via Kerberos, which the IMAP protocol supports.

3.4 Discussion

The Mona Internet transformations present to the user and application developer a file system interface to the Internet. By allowing remote resources to be manipulated as though they were part of the local file system, we enable several practical advantages that immediately have an effect on a user’s everyday tasks. As we argue later, the impact of the local mapping of remote resources can also have a dramatic effect on
the application developer, both in terms of development time and in the simplification and consolidation of the network programming interfaces.

We do not only envision browsing Internet resources as files using ls, cd, cat, etc. We believe that applications would still be used, but as front ends to the file system. These front ends would most likely have supporting programs or code, such as the mountftp and cleanftp programs discussed above.

From the user's perspective, the Mona Internet transformations simplify the manipulation of remote resources by allowing access to the resources via standard programs. This has an immediate effect on all kinds of users. For the beginner or first-time user of Unix, learning the arcane interfaces to FTP or NNTP, for example, can be nontrivial. Mona simplifies this task by allowing this type of user to learn just one interface, the file system interface. The user can then issue standard Unix commands like cd, ls, and cp, to traverse and manipulate remote resources. The more advanced user can similarly benefit from these transformations. Since remote resources act as though they are part of the local file system, these resources can be manipulated using powerful tools like grep, awk, and sed, to name a few.

While the value of the Mona Internet transformations to users is significant, they are at least as valuable to the application developer. Both the network and various protocol implementations are abstracted away and hidden behind standard file system system calls (open, read, write, etc.) The advantages for application developers using this model are numerous: easy access to APIs, faster development time, fewer bugs, etc.

Furthermore, when a new Internet resource (with a new access protocol) appears, existing programs can access the new resources without requiring any modification.

Programmers may never need to know the underlying details of either the protocols' implementation or the networking architecture. They may simply create transformation links, as shown earlier, and manipulate the links as local files. The transfer of data between local and remote machines is seamless, allowing the programmer to focus on features that differentiate their application. This is far better than the current model, which requires a programmer who wants to write an NNTP or IMAP client to either learn the protocol or find a library that implements the protocol. The process can be arduous and time consuming, which is especially costly for programs that use remote resources sparingly. By using the file system interface, the programmer is using a familiar service and is more likely to both complete the project faster and to make fewer mistakes.

Working in the Mona file system has further advantages because all of the other Mona transformations are available. Suppose one regularly accesses a compressed file on an FTP server. For example, consider the file foo.gz on a remote FTP server. A transformation link that applies both the FTP transformation and a decompress transformation can provide direct access to the decompressed data in the remote file. Not only is the remote file seamlessly transferred to the local disk, the user no longer needs to explicitly process the file to view its contents.

4 Results

The performance of tools that depend on remote resources are primarily a function of the latency of the connection between client and host. Furthermore, this latency dominates the local processing costs. For example, [WVS+S99] discusses a trace of real web accesses conducted a one week period at two large institutions. This study shows that the median HTTP access is approximately 400 ms and the average is greater than 1.7 seconds. In such an environment, the overhead of Mona is not significant. Therefore, there is little difference in performance between Mona Internet transformations and other methods when accessing remote servers.

Figure 2 shows the time to transfer files of different sizes from an FTP server located on the local area network. The time and variance increases with the distance to the FTP server. One curve shows the time to open a remote file that is guarded by the Mona FTP transformation, ftp.xform. On open, ftp.xform fetches the remote file and stores it on the local disk. The other curve shows the time to fetch a remote file and store it locally using a user-space program. The core FTP fetching routines in the two tests case are the same. The smallest file size displayed the largest performance difference, 15 milliseconds in the worst case.
The impact of network latency increases as file size increases, making accurate performance measurements of large file transfers difficult for both implementations.

It is useful to determine the overhead of implementing an operation as a user-level transformation. The open system call increase by a factor of 7 to 8 when a file is guarded by a user-level transformation. On an open, the kernel makes an upcall to the Mona daemon, which forks a child helper process. The overhead stated above does not include operations that occur in an open transformation, if any. In the FTP transformation, for example, the open transformation will make one or more FTP requests. In this case, the cost of open increases by many orders of magnitude. This is not particularly surprising, since these operations are not directly comparable. More details can be found in [Ken98, SKMF00].

The overhead of read and write system calls increase by a factor of 20 for the nop transformation executed in the kernel versus in user-space. Because the nop transformation is so trivial this represents the additional cost of going to user-space. When compared to kernel-resident transformations that perform equivalent operations, the user-space transformation has much lower relative overhead. Because the number of read and write calls is proportional to the file size, the absolute cost of transformations grows with the size of the file. Again, more details can be found in [Ken98, SKMF00].

Mona transformations compare favorably with conventional command line pipes. Figure 3 shows a
comparison of the two. Due to the overhead of establishing the transformation network, user transformation performance is worse than that of pipes until the file size approaches 256K, although only by a little over a tenth of a second in the worse case. Because Mona perform better during computation is outperforms pipes on larger files. For file sizes larger than 256K, Mona user-level transformations perform better than their pipe counterparts by as much as 66%.

It is also useful to determine the amount of space the Mona Internet transformations occupy. For each remote file that is mapped locally, two inodes are used, one to store the transformation link and another to store the base file. Since the transformation link is simply a symbolic link, it can never occupy more than 1K and no less than one file block. The base file, on the other hand, will take up as much space as the remote file to which it is mapped.

In both of these cases the functionality of the Mona Internet transformations outweighs the additional cost. The round trip time from request to receive using an Internet protocol is usually the dominant cost. Therefore, the overhead that Mona user-level transformations impose is small compared to the inherent latency of the Internet. Additionally, given the cost and capacity of disk storage, the amount of space that the Mona Internet transformations consume in the file system is unlikely to pose a problem.

5 Conclusion

The Mona Internet transformations make remote resources more accessible to both the user and the programmer. By providing a sensible mapping of several protocols onto the local file system, we enable the development of simplified applications and significantly enhance the utility of existing programs. All of this is done without losing the functionality of the protocols we implement.

While previous systems have implemented similar mappings for a unique protocol, Mona provides the foundation to implement many protocols. Furthermore, it allows all these protocols to be simultaneously present in the file system. The four protocols we implemented in this paper are by no means the only ones that can be implemented, and we expect additional protocols to bring us even closer to creating a comprehensive Internet file system.

We argue for raising the level of the file system, which raises the level at which application programming begins. By starting at a higher level, application programmers can get further faster. This approach is better than using equivalent functionality in a library because our system raises the level of all applications and tools, not just those which use the library.

References


