Ubiquitous Mirrors: Turning Clients Into Servers

Michael Dautermann
myke@cs.wayne.edu

Abstract: *Accessing software from web sites is very convienent and fast, that is, unless the site you're retrieving from is on the other side of the world or hindered by a very slow link. Mirrors have been one solution, but up until now the only easy way you could run your own mirror was to be Root on a UNIX (or OS-X) box. This paper will describe the problems associated with distributing software, both as a developer and as a service provider (i.e. a web site containing commercial or shareware software). This paper will also offer different ideas for solutions: making use of underutilized CPU and network bandwidth. Other forms of Distributed Software Distribution will be discussed, and we'll also propose a Software Archive Mirror application that runs on older Macintosh computers. Using data gathered from this particular package we'll be able to answer certain compelling academic questions such as: Does the physical location between client and server really matter? Or is it all about where you sit on the network?*

Introduction

For over 10 years now, I have been compiling and organizing a public domain and shareware library of files accessible via the Internet. A number of issues that have come up over these years that have affected accessiblity. This paper will attempt to address accessibility issues and problems solved by making use of a Distributed Systems process (or tool) commonly referred to as a "Mirror".

To the casual user browing on the Internet, a Mirror is a ftp server or website (usually relatively local to its audience) that is essentially a proxy between the customer and a Software Library (which we'll also call a "Source Site" or "Primary Download Site"). A Mirror contains files downloaded from one or more Software Libraries, and subsequently made available by the Mirror for public download. It's usually reliably and routinely updated (although between updates there certainly could be inconsistencies), and the number of users is significantly less than the number accessing the Primary Download Site. For example, if a software author released an update for a very popular game (like Quake or Bolo), the Primary Download Sites the author mailed the file would duly update their files. As soon as the game's fans find out about the update, these sites could quickly be swamped with requests to download the file.

If the software author waits to publicize the availability of the file about 24 hours after the file is made available, by then Mirror sites of the primary download site will have picked up the file, and the software author can include a list of maybe 27 additional locations around the world where game players can pick up the software update. For the software provider and the download site, this is accomplished with absolutely no effort, only patience.

To set up a Mirror, one would need a machine capable of running automated Perl scripts, access control over the FTP or web server daemon (to keep users in directories they're allowed to be in), and a relatively high network bandwith. There have been some new alternatives released in the recent past and we'll talk about them later in this paper.

Why should we (Software Authors & regular people, also known as users) care about Mirrors? What are typical problems?

To distribute software, a typical software author could choose a number of different options:

1) They could write "updating" code directly into the application. That is, the user clicks on a "check for update" choice in the menu and then the software goes out to ask a single server whether or not an update is available. The disadvantages of this approach are that the server could be down, the customer might be not on the net or behind a firewall, and then the customer actually has to get the software on their machine in the first place. The work for putting this functionality into an application might also be problematic.

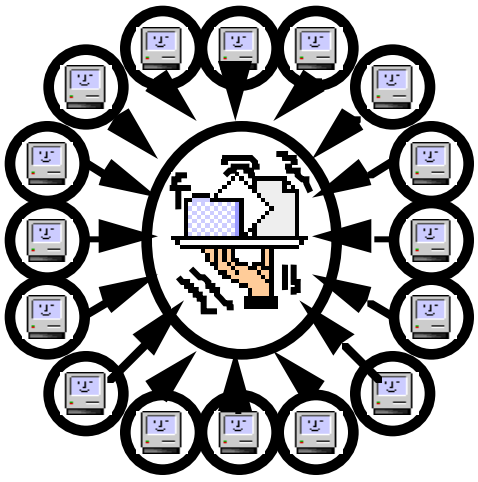
2) The Operating System could handle the updating. For example, the Macintosh OS has a nifty feature in the Control Panels folder called "Software Update". Getting ones software registered and placed into in this Software Update listing might be difficult to do though (as with many new techologies Apple releases, the documentation is very complex), and then we also have to get the software onto the machine to begin with (Apple's "Software Update" is aimed at commercial distributions).

3) A software author may choose to make his/her files available on a single web page that they own. The difficulties with this are that users will have to make do at least one hop to get to the software, usually through a Search Engine such as Google or through a "fake" Software Archive Site such as www.pure-mac.com; that is, a site that only lists the official shareware download pages and doesn't store any files locally. Customers could also "bookmark" this page in their Browser application, which may or may not be likely depending on how lazy the user is or how lengthy the Bookmark list is on the customer's machine. Author supplied web page URLs (e.g. "`reallygreatsoftware.html`", "`reallygreatsofwareV2.html`") are also much more subject to change than established download sites where one can look up files by typing in a URL under a well-known directory hierarchy (e.g. `http://archive.umich.edu/~mac/util/compression`).

4) To reach a maximum amount of users effectively, experienced authors of shareware & freeware release their files to well known sites such as Info-Mac or download.com. The easiest way to do this is to e-mail one copy of the file to an established distribution address (such as "macgifts@info-mac.org"), which redistributes the software to a number of locations that have volunteered to be Primary Download Sites.

Lastly, while the paragraphs in this section were addressed to authors and users of shareware, freeware & public domain software, some of issues may also apply to the commercial sphere (although for the

Figure 1. The web server your popular software can be found on may be under siege (or swamped)



majority of commercial software, distribution is usually carried out through shrink-wrapped boxes in stores and mail-order catalogs, secure & paid download sites and much more generous and aggressive marketing which guides customers to the location that would generate the most revenue).

Why should a Software Provider care about Mirrors? What are typical problems?

Shareware and freeware archive locations are typically operations that run on a shoestring (or non-existent) budget, typically on not so state-of-the-art machines. Conversely, the Software Libraries might be commercial operations (such as "download.com" or "tucows.com") which are loaded down with ads and aren't easy to navigate. Either way, once a Software Library is established and popular, the operators of the site will have a number of problems to worry about.

1) As illustrated in Figure 1, the library's point of entry could be on a single machine that could be attacked, could crash, or most likely, be swamped.

2) The WWW server (or servers) point to files kept on a single file server. If the web server or the file server goes down, accessibility is interrupted.

3) Even if a Software Library is kept on multiple webservers on a single network (or multiple networks), traffic generated can swamp local networks these servers are located on.

These problems were solved with the solution we introduced on the first page. Enthusiasts and fans of a popular library site create "Mirrors" distributed across the Internet. While these sites certainly help to distribute the load, when it comes to retrieving updated versions and lists of library

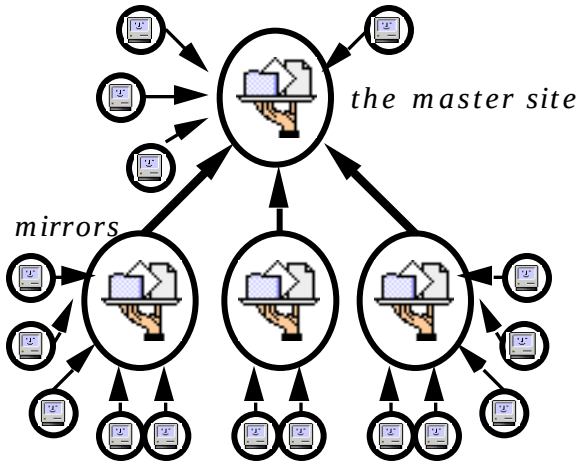


Figure 2. Mirrors distribute the load

files, mirrors were subject to the same accessibility problems (if the file server was down, if the network was swamped) as regular users.

Why are Mirrors an excellent solution?

Provided you have the right environment to start with, Mirrors should be very easy to set up and benefit from (as shown in Figure 2). Instructions on how to retrieve and install the Public Domain version of the Perl scripts that allow Mirroring can be found at `http://www.nottingham.ac.uk/pub/soar/setting-up-mirror.html`. While this package is an established tradition, we now turns our attention to a solution we would like to offer. Here are two solutions we'll propose:

1) Writing a public domain, open source application that can be run on older model Macintosh computers. This will allow for exceptionally easy Mirror creation (i.e. just install the software, do a quick configuration of the app, and let it go). It will also be politically easy (most academic and corporate departments have old computers lying around, it'll also be easier for enthusiasts to set up these mirrors anywhere they want... on campus, over DSL lines at home, in server closets, etc.). The way we'll architect

this software will also allow similar packages to be written for the Windows OS, or under Java; the idea being that enthusiasts and users without Root access (e.g. at home) may want to provide Mirror services.

2) We would like to glean performance data from these Mirrors. Using Browser Cookies and Traceroute data from selected downloads, we'll attempt to answer these questions: **How much does Physical Location matter in relation to file transfers? How much does Network Location matter?** This data will be handy not only to the research that is being done at the Master Archive location, but the person running the Mirror application will be able to view and easily interpret the throughput logs generated.

How about alternatives to Mirrors?

There are a number of options to be considered here.

One alternative to mirrors might be to incorporate "push" technology. The Master server would maintain a list of which updates are maintained on slave mirrors. This implies that the administration of the mirrors is either centralized (and corporate), or whoever owns the Mirror machines would have very little control or say in how often files are updated. This might not be so much of an issue if files aren't updated too often.

Another approach could be to "cache" files on Distributed Mirror sites, with TTL (time-to-live) expiration dates on files. A problem with this would be that files rarely updated on the Master file server would be routinely re-downloaded onto the Mirrors, and then once a file does get updated, it might take up to TTL time (a certain number of days) for the file to become fully propogated out to all the mirrors.

One viable new project called "FreeNet" (`http://freenet.sourceforge.net`) has a philosophy much like the World Wide Web, but with more of a Gnutella-like anonymous servers & clients sense. While Freenet offers huge availability potential, the drawbacks are that should a site that offers software choose to remain anonymous, the software author won't be able to make a contact in order to ask about updating or deleting old versions of their code (or worse, tracking down or removing cracked versions of the shareware).

How does a Library need to be setup in order to be Mirrored?

It doesn't really matter where the Master copy of the library lives. There is a library on the M.I.T. campus, the library can live on the Wayne State campus, the one I'm thinking of lives at U-M in Ann Arbor.

It also doesn't matter what is in the library. It needs to have compelling content (so as to attract customers from which we can compile good data). The library I have in mind (The Software Archives found at `www.umich.edu/~archive`) contains public domain & shareware software for Macintosh, for Microsoft Windows, for X/Motif Windows, etc.

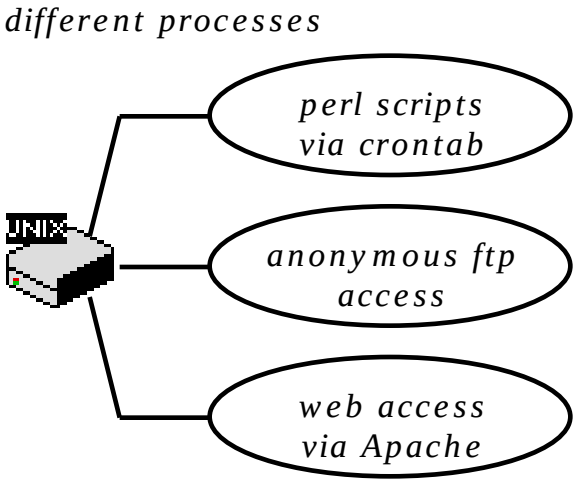


Figure 3. Mirrors are traditionally a collection of cooperating processes on a UNIX machine

To be mirrored most efficiently, the minimum thing that should be done is for a primary download site (the site the Mirror is retrieving files from) to regulary generate a recursive list of available files with timestamps. This can be easily done by putting a script with the command "ls -lR" into a crontab file.

When a Mirror machine connects to the server it's mirroring from to do a regular update, the first thing it does is retrieve the generated filelist. The Perl script then compares timestamps and additions/deletions to the filelist with what's found on the local filesystem. The script is then able to build a list of files that need to be syncronized between the Mirror and the remote master library.

How would our proposed Mirror package be different?

There are a number of limitations with the public domain "Mirror" package. As stated before, the currently available package depends on a number of processes working correctly on the server (see Figure 3). A person interested in administrating a Mirror should also be an experienced System Administrator, and they would also need to be "root" on the machine where the Mirror will be running from. Another limitation is that the mirror service usually has to be highly available (well connected, on a rack of machines in a server room) in order for it to be useful.

While implementations of Perl exist for other platforms beyond UNIX, robust FTP and HTML servers aren't necessarily up to the challenge of co-existing with running Perl processes.

One overall goal of this paper is to produce an application architecture which will allow the average enthusiast & computer user to set up his or her own Mirror site using underutilized machines (especially ones that would otherwise be turned off and gathering dust in a closet). In the application, we hope to combine the tasks of updating the local filesystem with an object oriented HTML (and possibly FTP) server. The code would be written in C++, although it would be architected so it could also be easily written in the more universal language of Java. A primary difference is that while a public domain Mirror package is dependent on a number of disparate processes running correctly and working together, our new Mirror application would be one main process, incorporating a main loop (using pre-MacOS X `WaitNextEvent` polling), user-enabled options and schedules. Each customer's file transfer would spawn off in a new thread, thereby keeping the application loop as uncluttered as possible.

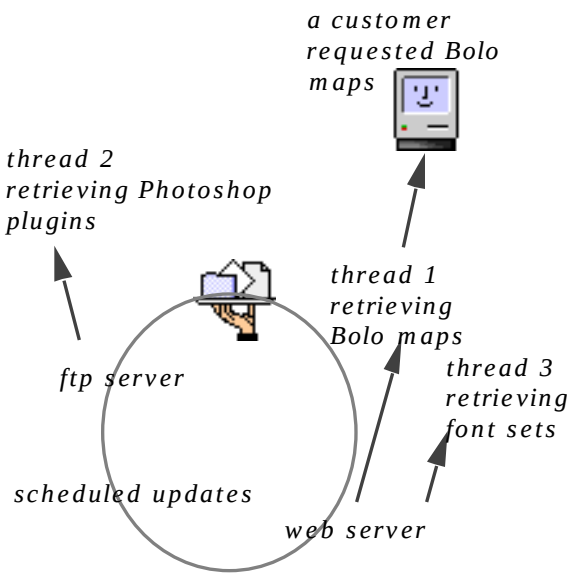


Figure 4. A Mirror app combines scheduling updates, with a ftp & web server

Because this application could be run at home or on a machine in a server closet, it would also might be desirable to add in access limits for certain times of the day. For example, a machine at a Mirror provider's home on a DSL/cable-modem

connection could provide more access during the day, when the owner is away at work. Machines at businesses may want to provide more access at night, when the local network isn't so congested. In either case, a machine that has hit user-defined limits can return the proper 400-series HTTP error ("4XX - Server too busy, try back later"). In fact, the error page returned could refer a user to another server that may be more available at that time (the links on this error page would be static, although creating a dynamic ally updated list of links would require programming effort and any algorithm might not be efficient under a machine that has hit system or user-defined limits).

Another benefit from our proposed package would be the possibility of *massive replication*; that is, while a typical software archive site might have a maximum of 30 or 40 machines dedicated to mirroring, our new package will enable anyone to set up a mirror and theoretically, we could see hundreds of mirrors of a popular web site using this software. If one mirror close to a user is down (due to a network or power outage), chances are very good another mirror will be nearby and available to download the files from.

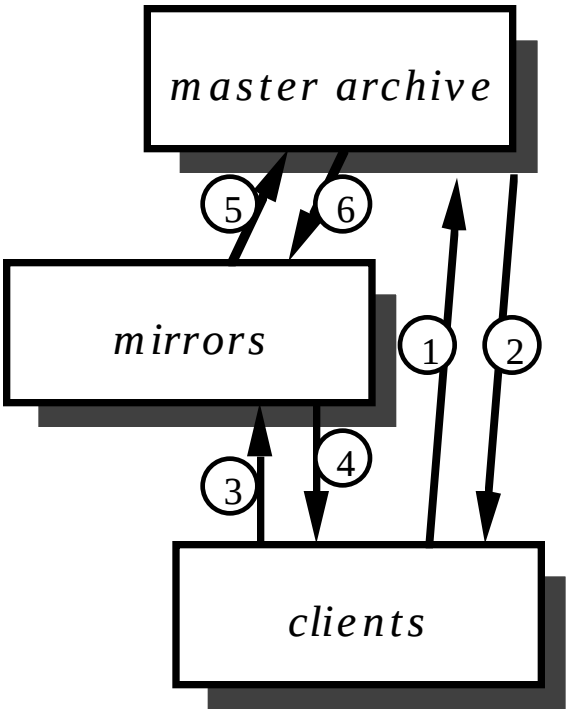


Figure 5. The order of events

Why and how our proposed Mirror application would use cookies?

If the Mirror provider chooses to mirror our preferred Master file server (the Software Archives at the University of Michigan), we will be able to generate data that will allow us to answer a number of useful questions.

The transaction process of a Mirror file transfer is illustrated in Figure 5. The first step in this process (step 1 in the figure) begins when an Archive user goes to the Software Archives main website in Michigan. While the user will have the ability to download software directly from the Master site, the site should offer up a list of mirrors that might be closer and faster (this would be step 2 in the figure). At the same time, the Master site will detect whether or not the user has been to the website before (by the existence of a cookie); if the cookie doesn't exist, the website will redirect the user to a page where the user will be asked for geographical location data. Users will also be given the option to opt-out, as well. All cookies will expire at the end of the research period (presently scheduled for December 1st, 2001).

If the user agrees to typing in geographic data, the cookie would probably only need two or three fields, and could be kept as a text type (so it could be easily viewed and verified from by the user). For example, a cookie on my machine would say:

```
MirrorUserCountry = US
MirrorUserCityState =
      Farmington Hills, MI
MirrorUserZip = 48336
```

Because of Security considerations by Browser developers, cookies can only be

read by the domains they're intended to be used by. For example, when a cookie is created, we must specify a domain of "cs.wayne.edu" when we want the cookie parameters to be sent back to a web server in the cs.wayne.edu domain. To create cookies for the Mirrors, the Software Archive will need to have a list of mirrors (which we'll generate manually, because no matter how optimistic and enthusiastic a Mirror administrator could be, a human would have to evaluate the Mirror site's availability; sometimes Mirrors are simply not reachable or reliable over the interconnected networks).

For each "blessed" (i.e.verified & approved) mirror application that we know is running our package, we will create a cookie that contains the parameters that we want to send back to that specific Mirror application.

When the customer accesses files through a mirror (steps 3 and 4 in Figure 5), our Mirror application would detect whether cookies are being sent back by a user. With these cookies, we'll be able to do two very useful things: log the data throughput for the file transfers, and do the equivalent of a network traceroute to determine the number of hops the data has to make to get to the user's location. While we can easily log the throughput of data on each file transfer, doing traceroutes over the network is somewhat more CPU intensive and the application should only do it on a random or proportional basis.

Each time the Mirror logs into the Software Archives to update and retrieve new files (listed as step 6 in the figure; on first use, this would obviously be step 1), it can deposit the log files into a "research results archive" and zero out its internal database & counters (listed as step 5 in the figure). With the data that is gathered by the distributed Mirrors, we'll be able to measure how efficiently a user was able to retrieve files from the distributed application.

For example, one can easily assume that a customer in Iowa can retrieve files faster from a Mirror also located in Iowa; but does the type of network the customer uses matter? Or is the transfer between the Iowa Mirror and the Iowa customer dependent and throttled by a link in another, remote state (such as New York)? As an example, Figure 6 shows an actual traceroute between a website in Michigan to another client in Michigan. The person downloading from Michigan probably would have gotten a faster download had s/he connected to a mirror in New York City.

```
terminator-myke:; traceroute 166.90.254.107
traceroute to 166.90.254.107 (166.90.254.107),
        30 hops max, 40 byte packets

 1  v-umce-rsug.c-arb4.umnet.umich.edu
 2  pc-arbrlks2.c-arb2.umnet.umich.edu
 3  atm3-0x8.michnet8.mich.net
 4  63-149-0-185.cust.qwest.net
 5  chi-core-01.inet.qwest.net
 6  jfk-core-02.inet.qwest.net
 7  jfk-brdr-01.inet.qwest.net
 8  pos1-1.core1.NewYork1.Level3.net
 9  so-4-0-0.mp1.NewYork1.level3.net
10  so-0-2-0.mp2.Detroit1.level3.net
11  gig9-1.hsa1.Detroit1.level3.net
12  166.90.248.22
13  es2-nrp7-atm0-0-0-s1.sfldmi.bullseyetelecom.net
14  ip166-90-254-107.sfldmi.bullseyetelecom.net
```

Figure 6. A Traceroute between a web server and a client

Many U.S. websites already point European users to European mirrors and vice versa. Doing random traceroutes like this may help us to programatically determine and suggest to a user that a Mirror in New York state would be faster to an Iowa customer than a mirror physically located in Iowa. These are examples of questions that would be interesting (and useful) to answer.