Enhancements to the Autofs Automounter

Ricardo Labiaga – Sun Microsystems, Inc.

ABSTRACT

An automounter is a system tool that enables administrators to share a uniform file system name space across an organization. Until the introduction of the enhanced autofs automounter, no automounter had integrated browsability of maps into the operating system. This paper describes enhancements made to the autofs automounter that enable entry listing in automounter maps without triggering mount storms. This allows applications to seamlessly browse potential mountable entries without the overhead of file system mounts. In addition, this paper describes the implementation of lazy mounting of hierarchies, which improves on-demand file system mounting.

Introduction

An automounter is a service that automatically mounts file systems upon reference and unmounts the file systems after a period of inactivity. This service has been traditionally used to access NFS file systems in large enterprise environments where centralized administration of the file system name space is preferred. An automounter will perform the mounts when a file system is first referenced without requiring the workstation user to acquire super-user access to perform the mount command. To the user, the mount of the newly accessed file system is transparent.

Three automounters are in wide use today. The first to become available was the automounter in SunOS 4.0. This automounter is available in many Unix system platforms, such as those from Auspex, HP, Compaq, IBM and SGI. A second automounter, the Amd automounter [8], is included in 4.4 BSD and has also been ported to many Unix systems. The third, the autofs automounter, was introduced in 1993 in Sun Solaris 2.3 and has been implemented on various other Unix system platforms since then, such as HP, IBM, Linux and SGI.

Both the SunOS 4.0 automounter and the Amd automounter are implemented as NFS servers [4]. The server is a daemon process, NFS mounted on directories that are required to be dynamically mounted. Each mount point is associated with a map that determines what components appear under the mount point and describes to which file systems they correspond. When the user process crosses the mount point, the kernel communicates with the daemon in the same manner it would communicate with any other NFS server. The daemon mounts the desired NFS file system on an alternate path and returns a symbolic link to this newly mounted file system. All lookups crossing the mount point are redirected by the kernel to the NFS server daemon, which returns a symbolic link to the mounted NFS file system.

The autofs automounter is considerably different than the traditional NFS server based automounter. It consists of three main components; the autofs file system, an auxiliary automount command utility and the automountd daemon.

The autofs file system is a virtual file system [6] (VFS) that intercepts requests to access directories that are not yet present. It calls the automountd daemon to mount the requested directory. The automountd daemon locates the requested path in the automounter maps, mounts the corresponding file system overlaying the autofs mount point, or mounts it on a subdirectory within the autofs file system. The subdirectory is created if necessary. Once the requested file system has been mounted, the original operation which accessed the autofs directory can proceed. Subsequent references to the mounted file system are redirected within the kernel by the autofs file system. No further intervention of the automountd daemon is required.

It is the responsibility of the automount auxiliary command to initially install the autofs mounts that connect the automounter maps into the file system name space. At system startup, the automount command reads the auto_master map and installs the initial set of autofs mounts into the file system name space.

```
# Master map for automounter
#
/net   -hosts      -nobrowse
/home  auto_home
```

Table 1: auto_master map.

Given the auto_master map listed in Table 1, the automount command installs the -hosts map on the /net directory and the auto_home map on the /home directory. The -hosts map is a special automounter map consisting of all available hosts in the network along with their respective exported file systems. Figure 1 illustrates the resulting autofs file systems. The following would be the equivalent autofs mounts:

```
$ mount -F autofs -o nobrowse \
                    -hosts /net
$ mount -F autofs auto_home /home
```

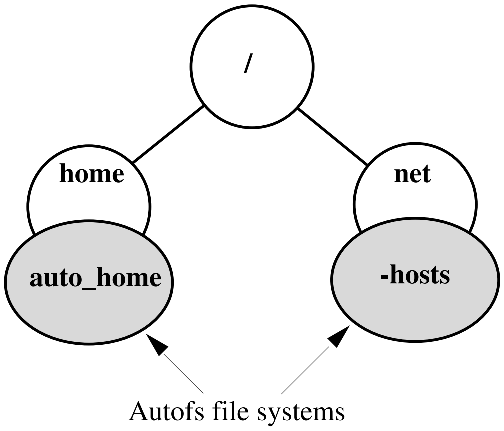

Figure 1: Autofs mount points.

By default the automountd daemon and the automount command will look for the maps in the /etc directory, unless otherwise specified in the auto_master map. The name service is consulted depending on the system configuration. On a Sun Solaris OS, the automount entry in /etc/nsswitch.conf determines whether or not the automounter will consult the name service directory. The current release of the Sun Solaris OS supports NIS and NIS+ [7] automounter maps. Support for automounter maps via the Lightweight Directory Access Protocol [5], will be added in a future release.

An autofs mount contains the following information:

- The name of the map associated with the mount point.
- The type of the map, either direct or indirect.
- The address of the user-level automountd daemon to contact for file system resolution.
- The default options to be used for mounting.

There are three kinds of automounter maps: direct, indirect, and executable. A direct map contains entries consisting of an absolute path, used as the key identifier, a corresponding file system to be mounted upon reference of the key and an optional set of mount flags to be used. Each direct map entry is itself installed as an autofs mount. Each one of these mounts constitutes a *direct mount*. A direct mount is defined as the autofs mount of an absolute path listed in a direct autofs map. With a direct mount, once the autofs mount point is accessed by a process, the autofs file system calls the automountd daemon providing its path and direct map name. The automountd daemon proceeds to mount the corresponding file system covering the autofs mount point. The autofs file system then redirects the blocked request to the newly mounted file system. Table 2 illustrates two direct map entries part of an auto_direct map.

```
/usr/dist -ro flash:/export/dist
/opt/onbld -ro flash:/export/onbld
```

Table 2: auto_direct map.

An indirect map contains entries consisting of a key identifier (a simple path component), a corresponding file system to be mounted upon reference of the key and an optional set of mount flags to be used. The indirect map itself is installed as the autofs mount, in contrast to direct maps whose entries are the actual autofs mounts. An *indirect mount* is defined as the autofs mount of an indirect map. With indirect mounts the autofs file system provides access to a directory of automatic mounts. Access to the autofs mount point itself does not trigger mounts, instead a request to lookup a subdirectory (key) in this directory will cause the autofs file system to call the automountd daemon with the subdirectory name and the map name. The daemon finds the name in the map, mounts the corresponding file system using the subdirectory as the mount point (not the autofs mount point) and replies to the autofs file system, which in turn redirects the blocked request to the new file system mounted. Table 3 illustrates an indirect map containing various home directory entries.

```
ashok   redback:/export/home/ashok
bev     turbo:/export/home/bev
brent   terra:/export/home/brent
david   jetsun:/export/home/david
warp    hp:/export/warp
peter   turbo:/export/home/peter
spencer austin:/export/home/spencer
```

Table 3: auto_home map.

An executable map is a special kind of indirect map. It is a local file with its execute bit set. The automountd daemon will execute the file and provide the name of the key to be looked up as an argument. The executable map returns the corresponding map entry on its stdout or no output if the entry cannot be determined. Refer to [3] for a detailed description of executable maps.

An extensive description of the features of the various automounters can be found in the references [1, 2, 3, 4, 8] at the end of this paper.

Limitations of the Autofs Automounter

Although the autofs automounter solved a number of problems seen in previous automounters [3], some limitations remained:

- There was inherent serialization in the autofs file system kernel component. This prevented concurrent mounts of entries that were part of the same automounter map.
- No support for browsability of maps was provided. A listing of an autofs directory only returned entries that had previously been mounted. Entries that had not yet been mounted were not listed.

- The mechanism used to mount and unmount hierarchies of file systems was prone to error. Temporary outage of a component lead to holes in the file system name space.

Architecture

In order to fix these limitations, the autofs automounter was re-architected for Sun Solaris 2.6. The new version remains split into three distinct components:

- The autofs file system, a kernel virtual file system that triggers all of the mounting and unmounting of file systems.
- The automountd daemon, a user level process responsible for performing the actual mounts and unmounts of the requested file systems.
- The automount command, a user level program that installs the initial autofs file system entry points.

The autofs file system and the automountd daemon communicate using connection-oriented RPC over the loopback transport. The RPC Protocol section describes this protocol in detail.

Browsability of Indirect Maps

Until the introduction of the enhanced autofs automounter, automounters lacked the ability to list all available entries in maps. A listing of an autofs directory referencing an indirect map only returned a list of directories already mounted, but did not list the entries which could potentially have been mounted, making it difficult for the end user to know what entries were available before they were first accessed.

This restriction was in place to prevent unintentional mounts of the entire map contents, a condition known as a *mount storm*. For instance an ls -l /home/* command, would have triggered a mount storm. This mount storm would have been caused by the *readdir*(3C)/*stat*(2) combination. The *readdir*(3C) makes the entire list of keys available to the browsing utility, such as *ls*(1) or a GUI file manager, which in turn issues a *stat*(2) of every entry to obtain its attributes. The *stat*(2) of an indirect map entry would have triggered the mount of the entry in order to verify its existence, and obtain its attributes. Performing this operation on every entry would have made directory browsing expensive and time consuming on relatively large maps.

The enhanced autofs automounter solves the mount storm problem by modifying the mount strategy. Previous automounters trigger the mount as soon as a new entry is looked up in an automounter map, so stat(/home/user) triggers the mount of the file system that corresponds to the user entry in the auto_home map. The enhanced autofs automounter postpones the mount of the file system until the indirect map entry is opened or a lookup of a component underneath it takes place. A simple *stat*(2) of the entry does not trigger a mount, instead the automountd daemon will simply query the map for existence of the entry.

Indirect maps may include a wildcard key “*”, which tells the automountd daemon that any key is valid in the map. In such cases, since not all entries are explicitly defined in the map, a listing of the map needs to be the combination of the dynamically created entries (those previously matched and mounted via the wildcard key) and the explicitly defined entries. This is achieved by first listing the entries that have already been mounted (either explicitly or via the wildcard) and then listing all explicitly defined entries in the map. A search for duplicates is done by the autofs file system before returning from the *getdents*(2) system call.

The first *getdents*(2) of an indirect mount point will return the entries currently mounted. Once all mounted entries have been listed, subsequent *getdents*(2) cause the autofs file system to send an AUTOFS_READDIR request to the automountd daemon to list the remaining entries in the map. This is done repeatedly until all entries in the map have been listed. The name of the map, the offset within the directory, and the number of bytes requested are included in the request. The automountd daemon will return the directory entries in chunks of the requested size, starting at the requested offset. Duplicates of entries already mounted are filtered out by the autofs file system. The maximum number of mounted entries listed is specified by the value of AUTOFS_DAEMONCOOKIE. This value is OS specific, though is usually in the order of tens of thousands of entries.

Consider the auto_home map listed in Table 3. Assume that the /home/brent and /home/peter directories have previously been accessed and mounted, and that our *readdir*(3C) library function generates *getdents*(2) requests of 25 bytes. A call to readdir(/home) will generate the following sequence of *getdents*(2) operations:

- The first getdents(/home, 25) lists the previously mounted entries brent and peter, and sets the next offset to AUTOFS_DAEMONCOOKIE to indicate it is done listing all currently mounted entries.
- Since the end of the directory has not yet been reached, a second getdents(/home, 25) is issued. Given that the current value of the offset is AUTOFS_DAEMONCOOKIE, the autofs readdir code handler issues a request to the automountd daemon, requesting the next 25 bytes in the directory. The automountd daemon in turn returns the first four entries in the map (22 bytes including termination characters) ashok, bev, brent, and david, and sets the next offset to AUTOFS_DAEMONCOOKIE + 4, since it returned four entries. It is the autofs file system that filters out the brent entry, since it is currently mounted and has already been listed.
- Again, since the end of directory has not yet been reached, another getdents(/home, 25) call is issued. The request is sent to the automountd

daemon again, which then returns the warp, peter, and spencer entries. The automountd daemon sets the end of directory boolean indicator in the structure containing the result, and replies to the autofs file system. The autofs file system filters out the peter entry since it is currently mounted, and *getdents*(2) returns the remaining two entries to the caller.

To maximize performance, and minimize kernel memory usage, no nodes are created as a result of a *getdents*(2) request. Nodes for the given entries will only be created if a process makes a subsequent lookup of entries listed by *getdents*(2) [i.e., ls -l].

These nodes are created with default attributes, which causes *stat*(2) to report the default attributes of nodes on which no file system has yet been mounted, instead of the directory attributes of the root of the mounted file system. Once the real mount is triggered, the attributes of the root of the mounted file system will be reported. There is no way to obtain the true attributes of the root of the file system without mounting the file system first. Remember that the intention is to be able to list the contents of an automounter map without having to mount the file systems referred to by the entries in the map. Note that this is the same behavior of direct map entries in previous automounters.

Performance

When an AUTOFS_READDIR request is received by the automountd daemon, it issues a request to the name service for the contents of the requested map. Since name services such as NIS and NIS+ do not support requests using byte ranges, the entire map needs to be requested on the first AUTOFS_READDIR call, even when only a portion of the map entries will be returned to the autofs file system at a time. Since the automountd daemon caches the entry listing, a subsequent AUTOFS_READDIR invocation will obtain its information from this cache. This leads to good performance of relatively large automounter map listings.

The directory listing of a 13,000 entry NIS automounter map (ls -f) takes approximately 15 seconds on a cold cache and two seconds on a warm cache. A long directory listing, which enumerates entries and their corresponding attributes (ls -l), takes approximately 40 seconds. A listing of a map with just under 150 entries is practically instantaneous. These informal measurements were obtained on a single CPU Sun Ultra 1 workstation running at 167 Mhz over a 10 Mb ethernet.

Optimizing Browsability of Maps

Information is more readily available if it is hierarchically organized. For example a large corporation may be divided into divisions and each division may be subdivided into organizations. Members of a team are easily identifiable when the division and organization to which they belong is known.

The autofs file system name space can be organized in a similar manner. It is easier to browse and administer multiple small automounter maps, than a single large flat map. Each automounter map can contain a combination of autofs and non-autofs mounts. For example, the company previously mentioned makes its employee home directories available through a hierarchical name space that mirrors its organizational structure. Its home directories are rooted at /home and the auto_master map contains the single entry:

```
/home   auto_home
```

The auto_home map contains:

```
legal   -fstype=autofs  auto_legal
eng     -fstype=autofs  auto_eng
HR      -fstype=autofs  auto_HR
mktg    -fstype=autofs  auto_mktg
CEO     flash:/export/olsen
```

The auto_eng map contains:

```
OS      -fstype=autofs  auto_OS
AMI     -fstype=autofs  auto_AMI
java    -fstype=autofs  auto_java
VP      flash:/export/smith
CTO     flash:/export/jackson
```

The auto_OS contains:

```
mct     iceberg:/export1/mct
spn     iceberg:/export1/spn
jim     flash:/export/jim
...
```

The user jim accesses his home directory as /home/eng/OS/jim. This results in the autofs mount of the auto_home, auto_eng, and auto_OS maps, as well as the NFS mount of flash:/export/jim. A directory listing of jim's organization simply returns the contents of the auto_OS map.

Potential Problems

Even though in practice mount storms are very rare, they can still occur. A recursive listing of entries under an indirect mount point can trigger the mount of all the file systems in the map. This may prove to be expensive and time consuming. It is conceivable that some legacy scripts and applications will need to be fixed to avoid this situation, especially those that do a depth-first search of the automounted file system at any point in order to build their own cache.

Disabling Browsability

Browsability may be disabled through the use of the -nobrowse map option for a specified map and children maps. The option is inherited. This option instructs the autofs file system not to list the contents of the specified map, causing it to simply return the listing of entries already mounted. This is similar to the old behavior, where only mounted entries were listed. The system administrator may want to specify this option in the rare event that a given application causes mount storms or when maps are unreasonably large, causing browsing to negatively affect system performance. This option only applies to the specified map and does not affect browsability on other areas of the file system name space. For instance, Table 1 shows the /net autofs mount point installed with the -nobrowse option. A *readdir*(3C) of the /net directory will only list the entries that have already been mounted. It will not list potentially mountable entries. The -browse option is used to re-enable browsability at any point in the autofs hierarchy. The default is -browse.

Incompatibility Issues

The -nobrowse and -browse options can not be parsed by older automounters. These options are not intended for inclusion in maps accessed via the shared name services, unless all automounters understand the new options or simply ignore unknown options. If an older automounter accesses an entry containing either of these options (or any other unknown option for that matter), it will cause the mount to fail since the older automounter cannot correctly parse the new options. The local system automounter map files are a better location for these options.

Improved Concurrency

The autofs automounter concurrency was severely limited when it came to accessing multiple automatic mounts under an indirect autofs mount.

In the case of multiple concurrent accesses to a single direct autofs directory, only one request needs to be issued to the automountd daemon to mount the corresponding file system. All other requests should block for the originating request to finish. This was achieved by flagging the autofs mount point with a ''mount in progress'' flag. Once the originating request received its reply from the automountd daemon, it signaled all other threads waiting on this mount point to proceed and clear the ''mount in progress'' flag. No new requests needed to be made to the automountd daemon since the mount had already been installed.

In the case of indirect mounts, the autofs directory was not the mount point of the newly mounted file system. Instead, the automountd daemon created a new mount point as a subdirectory of the autofs directory and mounted the new file system on top of it. At the time the request to the automountd daemon was generated, the autofs file system had no unique mount point to use as a synchronization point. Instead, it used the autofs indirect mount point to block other threads from generating a second mount request for the subdirectory. Unfortunately, this had the side effect of blocking lookups and mounts of other subdirectories under the same autofs indirect directory. This affected concurrency and lead to unnecessary hangs, where the lookup of a given path needed to wait for the mount of an unrelated path to be finished by the automountd daemon.

For instance, access to /home/warp first flagged the /home node with the ''mount in progress'' flag and then requested the automountd daemon to mount the corresponding file system on /home/warp. This is illustrated in Figure 2.

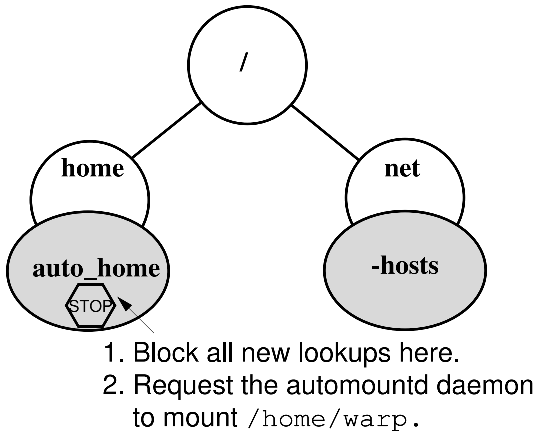

Figure 2: stat(/home/warp).

The automountd daemon in turn created the warp directory under the /home mount point and issued the mount system call to mount the new file system. If, at the same time, access to /home/bev was requested by a second process, the second process would unnecessarily block until the first process' access completed, since the /home node had previously been locked. Having a multithreaded automountd daemon was of no particular use at this point, since the request to mount /home/bev was blocked by the autofs file system. Figure 3 illustrates this scenario.

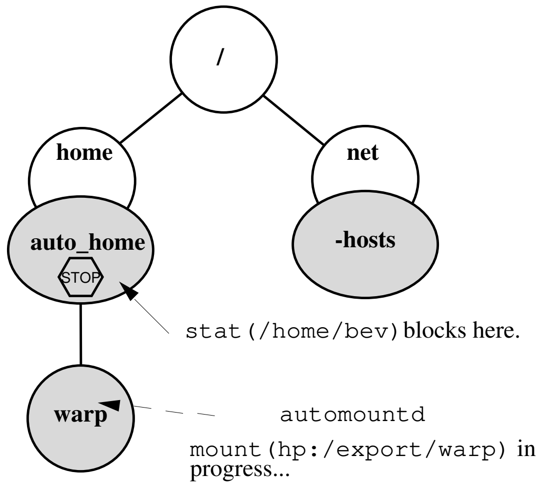

Figure 3: stat(/home/bev).

Once the mount of the new file system on /home/warp finished, the second request to install the /home/bev mount could be made to the automountd daemon.

To fix this major limitation, the enhanced autofs automounter moves the creation of the mount point from the automountd daemon to the autofs file system. For instance, upon lookup of /home/warp, the autofs file system determines that warp has not previously been accessed, creates the node for the name, warp, flags it with a "lookup in progress" flag, and issues a lookup request to the automountd daemon, as illustrated in Figure 4.

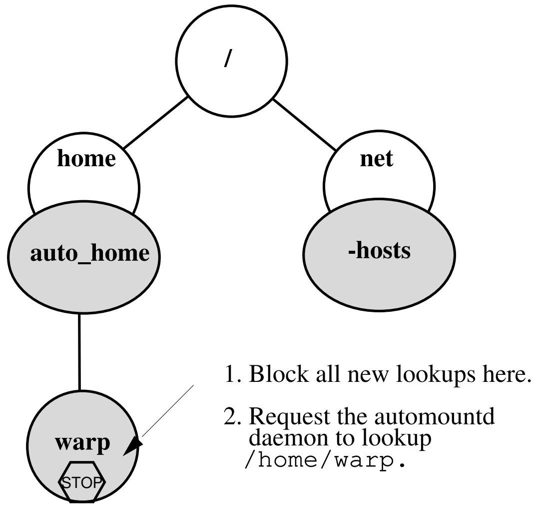

Figure 4: Non-blocking stat(/home/warp).

The daemon looks up the warp entry in the map and returns either success or failure to the autofs file system. A concurrent lookup of /home/bev can proceed since the /home node is not locked. The autofs file system creates a new node for the name, bev, under /home, sets the "lookup in progress" flag on /home/bev and sends the request to the automountd daemon. This is illustrated in Figure 5.

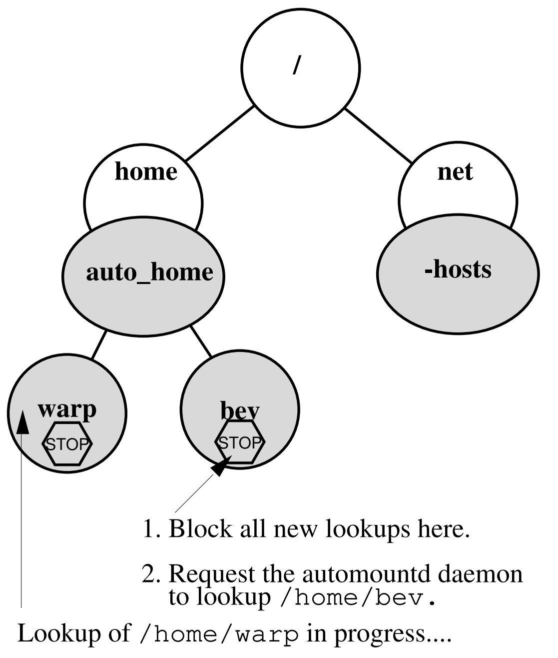

Figure 5: Concurrent stat(/home/bev).

At this time all new concurrent accesses to /home/bev will be blocked until the automountd daemon replies, since the bev node is flagged with "lookup in progress." This prevents flooding the automountd daemon with requests to perform duplicate lookup work. When the calling thread receives its reply from the automountd daemon, it will wake up any other threads waiting on the bev node, at which time they can return the same lookup status as the calling node obtained from its request to the automountd daemon.

Thus far, the nodes have only been looked up, no mounts have taken place. In order for the node to be covered with its corresponding file system, the node needs to be opened, or a lookup of a component underneath the node needs to take place. The RPC Protocolo Section describes the mount trigger policy along with the autofs RPC communication protocol in more detail. After the automountd daemon is done mounting the file system, the synchronization flags on the mount point are cleared and every process is allowed to proceed with its lookups and traverse into the newly mounted file system. This is illustrated in Figure 6.

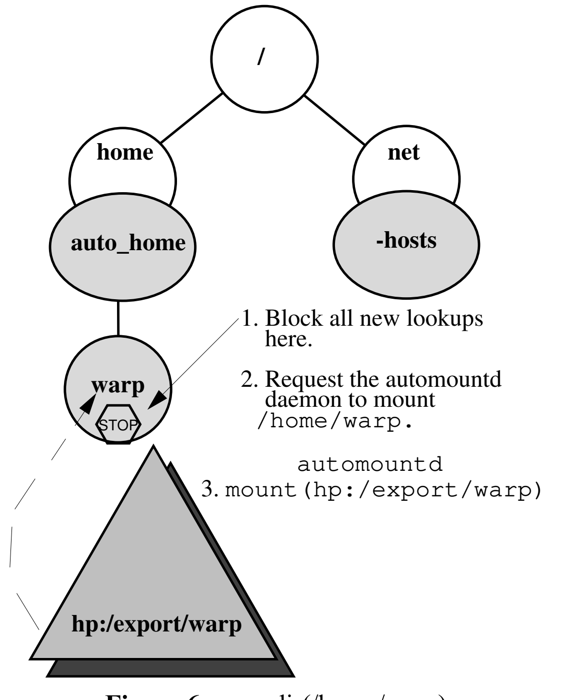

Figure 6: opendir(/home/warp).

Lazy Mounting of Hierarchies

Many times, more than a single file system needs to be mounted as part of a file system hierarchy. Existing automounters mount all members of this file system hierarchy at once, as soon as the top level file system is accessed. A common example of this is the use of the /net autofs mount. The automounter mounts all file systems from a given server when /net/host is referenced. These file systems may be hierarchically related, one within the other, and the entire hierarchy is mounted as a unit. As a side effect, the entire hierarchy of file systems needs to be unmounted as a unit as well. This presents a number of problems. First, a large number of file systems are mounted, even if only one is needed. Second, and a more significant problem, is that nested file systems have to be recursively unmounted and remounted if the entire hierarchy can not be unmounted. The remounting of elements is particularly prone to failure due to network discontinuity and time outs, which often result in a hole in the file system name space.

To address this problem, the enhanced autofs automounter mounts only the top level file system of a hierarchy on first reference, installing autofs trigger nodes where the next level file systems will need to be mounted upon reference. This provides better on-demand automounting of hierarchies by reducing the total number of mounts. The autofs trigger nodes are direct mount points. They trigger a request to mount the real file system when the trigger node is opened or a lookup of a component under this directory is performed.

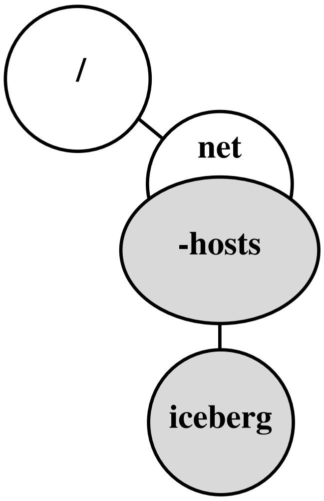

Figure 7: stat(/net/iceberg).

The automatic unmounting process is now reduced to unmounting the top level file system and its trigger nodes, instead of *n* file systems, multiple levels deep. If the top level file system is busy, only the trigger nodes are remounted, which is considerably faster than remounting other file systems, since the remount does not involve context switching in user space or over-the-wire communication with potentially dead or slow servers.

The following example describes the new mechanism for mounting hierarchical mounts in the enhanced autofs automounter. Assume the server iceberg exports the following eleven file systems:

```
/
/export1
/export1/home
/export2
/export3
...
/export9
```

/net is an autofs mount point that references the -hosts map.

As illustrated in Figure 7, a lookup of /net/iceberg creates the new node and makes a call to the automountd daemon requesting that it lookup the entry. If such entry exists, the new node is returned. If the entry does not exist, the node is removed and an ENOENT error is returned.

An opendir of /net/iceberg triggers a call to the automountd daemon to NFS mount iceberg:/, the top-level file system. After this top-level file system has been successfully mounted, the automountd daemon replies to the autofs file system indicating it needs to install nine new trigger nodes located at:

```
/net/iceberg/export1
/net/iceberg/export2
...
/net/iceberg/export9
```

Notice that only the trigger nodes for the level immediately following the top level were installed. This is illustrated in Figure 8.

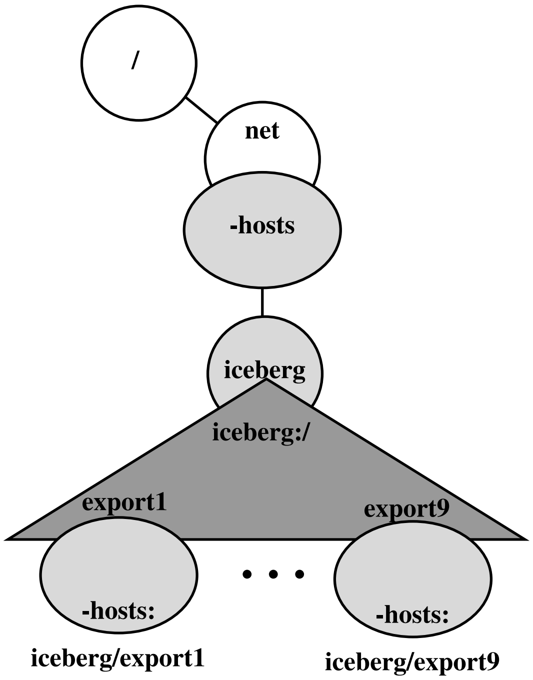

Figure 8: Top-level and trigger nodes.

If export1 is later referenced, the NFS file system iceberg:/export1 will be mounted, along with a trigger node for the next level of mounts, /export1/home in this case. This is illustrated in Figure 9. A subsequent access of /net/iceberg/export1/home would trigger the NFS mount of iceberg:/export1/home. Notice that no triggers under /net/iceberg/export1/home need to be installed, since the server does not share any file systems rooted deeper than iceberg:/export1/home.

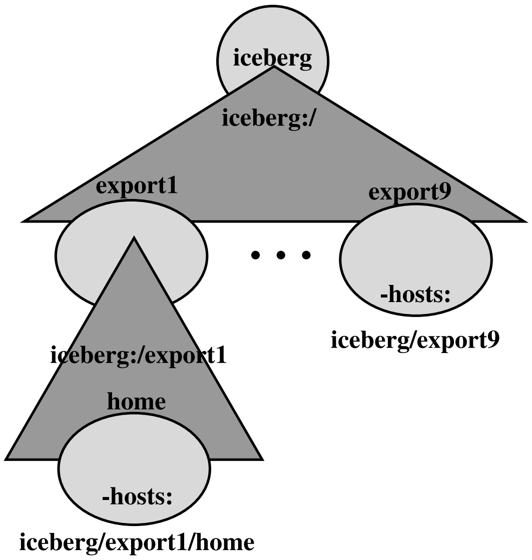

Figure 9: stat(/net/iceberg/export1/dir).

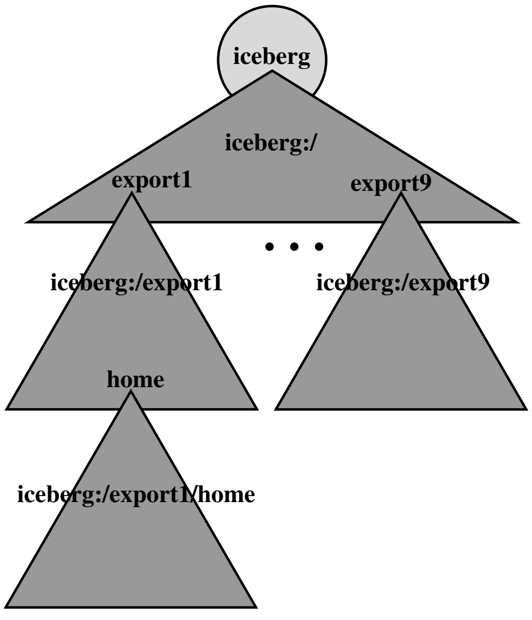

Figure 10: Older automounters mount all file systems on first reference.

There is no need to mount file systems that are never touched. Only the file systems accessed have been mounted along with the trigger nodes that enable the mounts of the next level. In contrast, the older automounters would have mounted the entire hierarchy on first reference of /net/iceberg as depicted in Figure 10.

Unmounting

The automatic unmount of a hierarchy is performed depth-first. A file system and its trigger nodes are unmounted as a unit. In the example depicted in Figure 9, the iceberg:/export1 file system and the iceberg/export1/home trigger node are both unmounted at the same time. This leaves the iceberg/export1 trigger node exposed, ready to trigger a new mount if a future access takes place. The unmount of iceberg:/ will not be attempted as long as any of its trigger nodes are covered by a mounted file system.

The RPC Protocol

The autofs file system and the automountd daemon communicate using connection-oriented RPC over the loopback transport. The communication protocol has been extended to six basic operations. All RPC calls are initiated by the autofs file system, they wait in a listening state until the automountd daemon replies or the calling thread is interrupted. The autofs file system will initiate a call to the automountd daemon when a thread references a trigger autonode. The six basic operations follow:

AUTOFS_LOOKUP

Description: Performs a lookup of a key in a map. This request is triggered when a thread performs a lookup of a not-yet mounted key in an indirect autofs mount.

Arguments: The request takes as argument the name of the map, the key being looked up, and the directory path.

Results: The automountd daemon returns success if the entry exists in the map or an error if it does not exist.

AUTOFS_MOUNT

Description: Performs a mount of a key from a map at the given mount point using the specified options. This operation is triggered under the following conditions:

- Direct autofs mounts: Triggered when a thread opens a direct autofs mount or performs a lookup of a component of the direct autofs mount.
- Indirect autofs mounts: Triggered when a thread opens a key of an indirect autofs mount (i.e., opendir(/home/warp)) or performs a lookup of a component below the key of an indirect autofs mount (i.e., stat(/home/warp/.cshrc)).

The automountd daemon matches the corresponding entry in the map and performs the required mount. For hierarchical mounts, the automountd daemon builds a list of structures corresponding to the next level mounts, each containing the necessary information to perform an autofs mount where a new file system mount needs to be triggered. These are used by the autofs file system for installation of the next-level trigger nodes. Certain types of map entries do not require a top-level mount. For these entries, the automountd daemon does not perform any mounts itself and simply returns a list of mount structures to be installed by the autofs file system as trigger nodes.

Arguments: The request takes as argument the name of the map, the key being mounted, the mount point, the default map options, and the type of the map.

Results: The automountd daemon returns a status code indicating the result of the operation and any necessary trigger nodes that need to be installed by the autofs file system.

AUTOFS_POSTMOUNT

Description: Adds a mount entry to etc/mnttab. Many Unix implementations maintain the mount table in user space (/etc/mnttab). To facilitate access to this table, an AUTOFS_POSTMOUNT operation is provided. A request is sent to the automountd daemon to append a path to /etc/mnttab after an autofs in-kernel mount has been performed. This is not necessary for OS implementations that maintain an in-kernel mount table.

Arguments: The request takes as argument a list of special devices, their mount point, file system type, mount options, and device identifiers to be added to the mount table.

Results: The automountd daemon returns a status code indicating success or failure.

AUTOFS_UNMOUNT

Description: Unmount the path corresponding to the specified device identifier.

Arguments: The request takes as argument the device identifier of the file system to unmount.

Results: The automountd daemon returns a status code indicating success or failure.

AUTOFS_POSTUNMOUNT

Description: Remove the path from /etc/mnttab corresponding to the specified device identifier. This is used to complete unmounts performed in the kernel (not via the *umount*(2) system call). Not required for OS implementations that maintain an in-kernel mount table.

Arguments: The request takes as argument a list of device identifiers to be removed.

Results: The automountd daemon returns a status code indicating success or failure.

AUTOFS_READDIR

Description: Requests a list of entries in a map. The automountd daemon reads these entries from the specified map and returns an array of *dirent*(4) structures beginning at the specified offset.

Arguments: The request takes as argument the name of the map, the starting offset and the total number of bytes requested.

Results: The automountd daemon returns an array of *dirent*(4) structures, the total size of the entries read, the last offset in the list, and an end of directory indicator.

Availability

The enhanced autofs automounter is available in Sun Solaris 2.6 and Solaris 7. The source is available to all ONC+ licensees, which include many major Unix vendors.

Future Work

An important limitation of automounters today (including the enhanced autofs automounter described on this paper) is that changes to the automounter maps are not effective immediately if the modified map entry has previously been mounted. For instance, if a given server shares a new file system after a client has already mounted a previously shared file system, the client will not have access to the new information. The newly exported file system will become visible to the client after /net/server and its trigger nodes are automatically unmounted after a period of inactivity on the client and mounted again on subsequent access. This is problematic to the user since newly exported information can not be readily accessed on clients.

Clearly, one solution to this problem is to provide some kind of mechanism that notifies the autofs file system that the mounted hierarchy has changed. However, not all directory name services provide time stamp capabilities, which would make it difficult for the automounter to determine when the maps have been updated. Some of the automounter maps are dynamic, such a -hosts. It is not practical to continuously poll servers to see if their list of shared file systems has changed. One solution would be a mechanism to manually notify the autofs file system with a list of mount points that need to be refreshed. A new option to the automount command specifying the path that needs to be refreshed may suffice.

Acknowledgments

The author would like to thank his colleagues from Sun Microsystems, Inc. who have provided valuable ideas and discussion material for the enhancement of the autofs automounter, in particular Ashok Advani, Brent Callaghan, David Robinson, and Peter Staubach.

Author Information

Ricardo Labiaga is a Member of Technical Staff at Sun Microsystems, Inc. He holds a Bachelors of Science degree in Computer Science and a Masters of Science in Computer Engineering from The University of Texas at El Paso. For six years, he has worked on a variety of projects within the Network File Systems Group at Sun Microsystems, Inc., with a primary focus on automounting. Reach him electronically at <labiaga@eng.sun.com>.

References

[1] Brent Callaghan, "The Automounter – Solaris 2.0 and Beyond," *1992 Sun User Group Conference Proceedings*.

[2] Brent Callaghan, "The Automounter – Using it Effectively," *1990 Sun User Group Conference Proceedings*.

[3] Brent Callaghan, Satinder Singh, ‘‘The Autofs Automounter,’’ *Summer 1993 Usenix Conference Proceedings*.
[4] Brent Callaghan, Tom Lyon, ‘‘The Automounter,’’ *Winter 1989 Usenix Conference Proceedings*.
[5] Yeong, W., Howes, T., and S. Kille, ‘‘Lightweight Directory Access Protocol,’’ *RFC 1777*, March 1995.
[6] S. R. Kleiman, ‘‘Vnodes: An Architecture for Multiple File System Types in Sun UNIX,’’ *Summer 1986 Usenix Conference Proceedings*.
[7] Chuck McManis, ‘‘Naming Systems: A Replacement for NIS,’’ September, 1991, *Sun UK User Group Conference Proceedings*.
[8] Jan-Simon Pendry, ‘‘Amd – An Automounter,’’ *Technical Report*, Department of Computing, Imperial College, London, England, 1989.