Regular Paper

GMount: Building Ad-hoc Distributed File Systems by GXP and SSHFS-MUX

Nan Dun,[†1] Kenjiro Taura[†1] and Akinori Yonezawa [†1]

GMount is a file sharing utility built by using GXP and SSHFS-MUX for the wide-area Grid environments. By GMount, non-privilege users are able to build their own ad-hoc distributed file systems in seconds. The file lookup in GMount can be efficient in terms of communication cost if the network topology is given. In this paper, we present the design and implementation of GMount, as well as its evaluation on a large scale Grid platform — InTrigger.

1. Introduction

Network file system[1] and conventional distributed file systems[2)–5)] provide data sharing approaches for parallel data processing in cluster or Grid environments. They are usually deployed in advance by system administrators and non-privilege users can hardly change the system configurations to meet their demand for specific applications. For example, users are unable to add or remove storage nodes, or to change the previously assigned global namespace.

GMount is an ad-hoc distributed file system built by using GXP and SSHFS-MUX. GMount works in userspace and any user can easily create his/her own distributed file systems on arbitrary nodes across multiple clusters in seconds. Since it uses SSH as communication channel, GMount is adaptable in NAT or firewall conditions. If network topology is available, GMount becomes more efficient in terms of communication cost because it utilizes this information to make file lookup behave in a locality-aware manner.

2. Building Blocks

2.1 GXP

GXP (Grid and Cluster Shell)[6)] is a parallel shell that allows user interactively execute commands on many remote machines across multiple clusters. It provides a master-worker style parallel processing framework by which users can quickly develop scalable parallel programs. We employed this framework as GMount's workflow and let GXP be the front-end via which users can manipulate GMount.

†1 Graduate School of Information Science and Technology, the University of Tokyo

2.2 SSHFS-MUX

2.2.1 Overview

SSHFS (Secure Shell FileSystem)[7)] enables users to seamlessly manipulate files on remote machines as local ones via SSH channel. SSHFS is implemented by using FUSE (Filesystem in Userspace)[8)] and SFTP (Secure File Transfer Protocol)[9)]. Since recent Linux kernel (version >= 2.6.14) includes FUSE by default, the installation or configuration of SSHFS does not need superuser privilege.

SSHFS-MUX (SSHFS Mutiplex) is an extension of SSHFS. It includes all merits of SSHFS as well as extra features that make it more convenient for users to handle with multiple remote machines. The difference between SSHFS and SSHFS-MUX can be intuitively illustrated by their usages shown in **Fig. 1**.

```
A$ sshfs B:/data /mnt/B
A$ sshfs C:/data /mnt/C
A$ unionfs /mnt/B /mnt/C /mnt/all
```

(a) SSHFS with UnionFS

```
A$ sshfsm B:/data C:/data /mnt/all
```

(b) SSHFS-MUX Merge Operation

```
B$ sshfsm C:/data /inter
A$ sshfsm B:/inter /mnt
```

(c) SSHFS-MUX Relay Operation

Fig. 1 Usages of SSHFS and SSHFS-MUX

SSHFS can mount only one host to a mount point once, or otherwise combinely using UnionFS[10)] as in **Fig. 1(a)**. Instead, mounting multiple hosts to the same mount point at a time is common in SSHFS-MUX, which is referred as *Op-Merge* in **Fig. 1(b)**. In Op-Merge, mount target appears later in arguments will be first checked on file request. For example

in Fig. 1(b), a file lookup request destined to `/mnt/all` will be redirected to branch `C:/data` first. Only when the target file is not found in `C:/data`, the request will go next to branch `B:/data`.

Another usage scenario of SSHFS-MUX is called *Op-Relay* as shown in **Fig. 1(c)**. It suggests that SSHFS-MUX can mount target host indirectly via intermediate mount point on which the target host has been mounted. There are two reasons to do so: 1) The load of target host serving many SFTP clients can be migrated to intermediate hosts; 2) If target host is behind NAT router or firewall, we can take advantage of gateway as intermediate node and let it relay the mount. But meanwhile, *Op-Relay* will introduce overheads, such as context-switch in FUSE and potential network delay, that need to be taken into consideration.

2.2.2 SSHFS + Union File System

SSHFS-MUX can be replaced by the combination of SSHFS and unionfs (union file system) to implement the same functionality, but it provides advantages over SSHFS+unionfs in following aspects.

First, SSHFS-MUX is easy to use. It can save users creating intermediate mount points for SSHFS branches merged by UnionFS. Besides, kernel space implementation of union file system (e.g., UnionFS[10]) requires additional privilege to perform mount operation.

Second, SSHFS-MUX has better performance. To compare the I/O performance of SSHFS+unionfs and SSHFS-MUX, we let them mount *localhost* to remove the network latency and perform read/write tests under the same condition. In this experiment, we chose UnionFS[10] as the test candidate for kernel space implementation of union file system and UnionFS-FUSE[11] for user space one.

Figure 2 shows the results. Here, SSHFS-MUX is slightly lower than SSHFS in read performance and has the best write performance. SSHFS+unionfs-FUSE has the lowest overall performance since it needs to perform costly context switch twice through FUSE kernel module.

Third, SSHFS-MUX compacts the interaction among components, reduces the effort of deployment, and makes it easier to debug GMount.

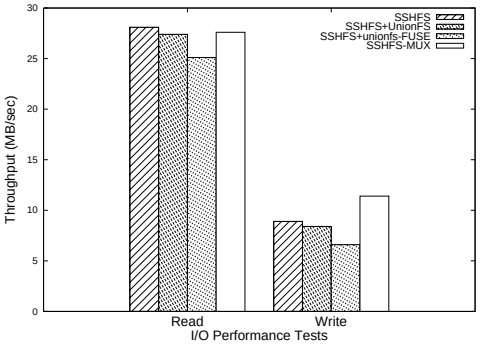


Fig. 2 I/O Performance Comparison between SSHFS+unionfs and SSHFS-MUX

3. Design Details

3.1 Overview

In GXP master-worker framework, the root node of GXP login spanning tree will become the *master* and all nodes are *workers*. **Figure 3** demonstrates the three stages of GMount execution.

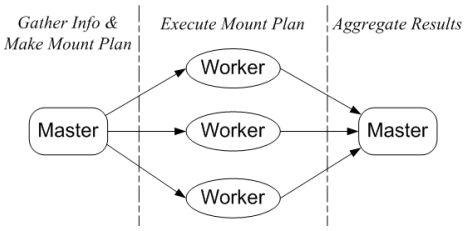


Fig. 3 GMount Workflow

Planning Stage Master first gathers the workers information such as total nodes number, hostname, ip address, etc, and generates a mount spanning tree (different from GXP spanning tree) based on specified algorithms. Then master broadcasts this tree structure along with mount flag to all of workers.

Executing Stage After receiving instructions from master, workers (include master itself) can determine their mount phase and mount targets according to their position in mount spanning tree. When every node is ready, all nodes synchronize their execution by using GXP barrier and then carry out proper operations (i.e., SSHFS-MUX Op-Merge and Op-Relay) according to predefined mount algorithms.

Aggregating Stage Each worker will send their exit status and error message, if any, back to master after mount phase is finished.

Finally, master aggregates the results and prompt user for success or failure of the execution of GMount.

Figure 4 shows the usage of GMount. First, user selects as many nodes as they like by GXP. Then user can specify GMount actions (e.g., "aa" for all-mount-all and "u" for umount) with arguments in GXP mw command.

```
$ gxpc explore node[[000-010]].cluster.net
$ gxpc mw gmnt -a aa /share /mnt
$ gxpc mw gmnt -a u
```

Fig. 4 Usages of GMount

In order to facilitate the description, the details of mount phase will be first given in 3.2, and the GMount spanning tree construction algorithms will be discussed in 3.3.

3.2 Mount Phase

There are four kinds of mount phase in GMount: one-mounts-all, all-mount-one, all-mount-all, and umount.

Throughout this section, we use a simple spanning tree as shown in **Fig. 5** to illustrate the details of executions of SSHFS-MUX on each host. We assume that there are three directories for each host: sharing directory `/share`, mount point `/mnt`, and intermediate directory `/inter`.

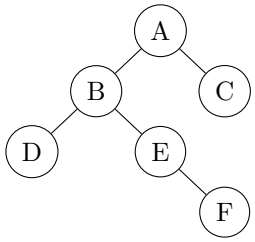


Fig. 5 Examplary Spanning Tree

3.2.1 One-Mounts-All Phase

Figure 6 shows the algorithms and **Figure 7** shows the executions in one-mounts-all phase. After one-mounts-all phase, root node is able to see all contents in `/share` of all nodes via its `/mnt` directory.

3.2.2 All-Mount-One Phase

Figure 8 shows the algorithms and **Figure 9** shows the executions in all-mount-one phase. After all-mount-one phase, every node will see all contents in `/share` of root node via its `/mnt`.

3.2.3 All-Mount-All Phase

Accordingly, the goal of all-mount-all is to

```
OneMountsAll
01 if self is leaf then
02   noop
03 else:
04   if self is root then
05     mount_point ← "/mnt"
06   else
07     mount_point ← "/inter"
08   targets ← φ
09   t ← self : /share
10   targets ← targets ∪ t
11   foreach child in self.children
13     if child is leaf then
12       t ← child : /share
15     else
12       t ← child : /inter
16     targets ← targets ∪ t
17   exec("sshfsm targets mount_point")
```

Fig. 6 Algorithms in One-Mounts-All Phase

```
A$ sshfsm B:/inter C:/share A:/share /mnt
B$ sshfsm E:/inter D:/share B:/share /inter
E$ sshfsm F:/share E:/share /inter
C, D, F$ No Operation
```

Fig. 7 Examplary Executions in One-Mounts-All Phase

```
AllMountOne
01 mount_point ← "/mnt"
02 targets ← φ
03 if self is root then
04   t ← self : /share
05   targets ← targets ∪ t
06 else:
07   if self.parent is root then
08     t ← self.parent : /share
09   else
10     t ← self.parent : /mnt
11   targets ← targets ∪ t
12 exec(sshfsm targets mount_point)
```

Fig. 8 Algorithms in All-Mount-One Phase

```
A, B, C$ sshfsm A:/share /mnt
D, E$ sshfsm B:/mnt /mnt
F$ sshfsm E:/mnt /mnt
```

Fig. 9 Operations in All-Mount-One Phase

enable every host to see all contents in sharing directories (`/share`) of all hosts via its mount point (`/mnt`).

Naive All-Mount-All If we combinely use one-mounts-all in Fig. 6 and all-mount-one in Fig. 8, then it is straight to have all nodes share their `/share` on their `/mnt`.

Clearly, in this simple mount scenario, all mount paths go through the root node and make the root a single bottleneck.

Scalability There will be a problem if we create only one spanning tree for many nodes of multiple clusters using above naive approach. A scalable solution is to construct a few spanning trees for each cluster and let the roots of these spanning trees form another sharing hierarchy in which the Grid-scope namespace is constructed.

Locality-Aware Lookup By using Op-Merge of SSHFS-MUX mentioned in 2.2.1, each node will put mount targets that are close to itself later in SSHFS-MUX execution command. Therefore, node are able to lookup files by sending request to affinity branch first. For example, when host A in Fig. 7 lookup files in `/mnt`, it will first lookup branch `A:/share`. If target file is found in `A:/share`, it will not send messages to other branches.

3.2.4 Umount Phase

In umount phase, GMount parallelly invokes `fusermount` to umount all SSHFS-MUX and performs cleanup jobs such as removing temporarily created intermediate mount points.

3.3 Spanning Tree Construction

Due to the overhead in Op-Relay of SSHFS-MUX, the mount spanning tree should be as fat as possible instead of tall.

Hostname Prefix Usually, nodes within one cluster share the same prefix of hostname. Therefore these nodes are grouped as candidates of one spanning tree and then perform following steps: 1) Randomly choose one node as root and start 2) at root; 2) Randomly choose K nodes to be children of the node; 3) For each newly added child, repeat 2) until all nodes are included in spanning tree.

Network Topology If physical network topology is available, GMount will create spanning tree as following: 1) Let the node closet to top switch to be the root of spanning tree and start 2) at root; 2) Choose first closet K nodes in terms of hops to be children of the node; 3) For each newly added child, repeat 2) until all nodes are grabbed into spanning tree.

4. Implementation and Availability

Though SSHFS-MUX is implemented in C language, GMount is written in Python as a module of GXP.

Both SSHFS-MUX and GMount are open source and are available at `http://www.yl.is.s.u-tokyo.ac.jp/~dunnan/sshfs-mux`, and GMount also comes with latest GXP release at `http://www.logos.t.u-tokyo.ac.jp/gxp`.

5. Known Problems

5.1 Limitations of SFTP

Both the functionality and performance of SSHFS-MUX highly depend on the evolution and implementation of SFTP (i.e., OpenSSH) because it uses SFTP as its underlying communication channel. Two major limitations of SFTP lead to non-trivial problems when using GMount: its partially POSIX-compliant and lack of automatic TCP buffer tuning.

5.1.1 Incompability with POSIX

Versions of SFTP older than 5 does not support overwriting rename[12] because rename operation is not atomic. SSHFS/SSHFS-MUX implemented a workaround by creating intermediate files on overwriting rename to solve this problem.

However, even with rename workaround, overwriting rename does not work in Op-Relay and hence in GMount. This is because hard link will not be available until version 6 of SFTP (latest OpenSSH is 5.0/5.0p1). SSHFS-MUX provides another workaround called "fakelink" for this problem by caching an alias for link target file internally and redirect all file requests on this alias to the original file.

5.1.2 SSH Receive Buffer

The OpenSSH[9] has a nature limitation to transfer bulk data in long-fat network environments due to its fixed receive window. This problem is examined in 13) and a patch of OpenSSH called HPN-SSH has been proposed. The support for HPN-SSH in SSHFS-MUX can be enabled at compile time.

5.2 Cache Inconsistency

Cache is ubiquitous in GMount. For each node, there are three places where file attributes and contents may be temporarily kept: 1) SSHFS-MUX cache; 2) FUSE cache; 3) Local system cache.

Currently, no cache consistency model is em-

ployed by GMount and some usage case may lead inconsistent views of shared files among clients. Users can disable cache in both SSHFS-MUX and FUSE by runtime options. The examination and solution of cache inconsistency problem will be proposed in future work.

5.3 Concurrent SSH Connections

In all-mount-one phase, non-leaf node may encounter simultaneous connection establish requests from its children, which indicates that SSH server on this node should allow to accept many concurrent SSH connections. Default SSH server may deny these concurrent connections, and it is configurable by modifying the value of `MaxStartups` ⋆1 in file `/etc/ssh/sshd_config` on server side.

6. Experiments and Evaluation

6.1 Experimental Environments

Our experimental environments, called In-Trigger, consists of 8 sites with over 200 nodes. The bandwidth and RTT between sites are shown in **Table 1**. We use Gfram[2] and HDFS (Hadoop File System)[14] as reference systems in our experiments.

6.2 Experimental Results

6.2.1 Filesystem Construction Time

To examine how filesystem construction time scales on nodes number, we incrementally append extra nodes to GMount and measure its mount/umount time. We added one entire site for each run in the first test and added only 4 nodes from the same site in the second run.

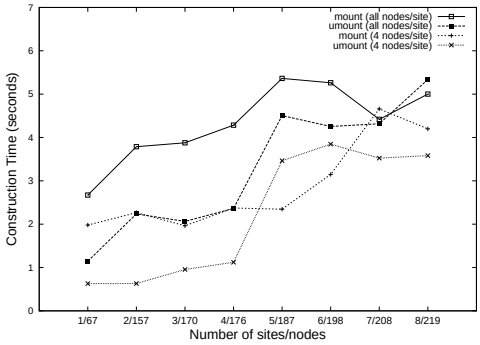


Fig. 10 GMount Construction Time

In **Fig. 10**, we can read that the filesystem construction time is basically related to network latency between sites. The number of nodes does not significantlly increase time of mount or umount.

6.2.2 I/O Performance

Since the write performance of distributed file system is highly related to the access pattern of clients, we will leave the evaluation of write in future and only present read performance here.

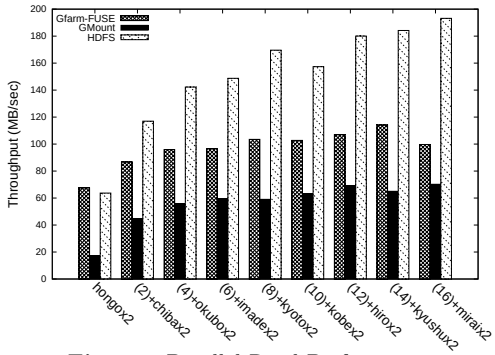


Fig. 11 Parallel Read Performance

In read test, we first create a large file (1GB) in one site. Then we increase 2 clients from different sites each time to do parallel reading from this file. The aggregate throughput of read is shown in **Fig. 11**. Each file system achieve higher throughput when clients number increases. GMount is the slowest because it uses SFTP channel and the transfer path is relayed by SSHFS-MUX. HDFS achieved the best because it breaks large files into trunks that can be utilized for parallel transfer.

6.2.3 Metadata Operation

We tested the performances of metadata operations when systems works in LAN and WAN.

Figure 12(a) shows the result in LAN environments. For Gfarm and HDFS, the operation time is proportional to the round trip time when they send request and get reply from metadata server, which is similar to NFS. GMount is slightly slower than Gfarm because its operations need to be relayed by a few nodes along the mount spanning tree.

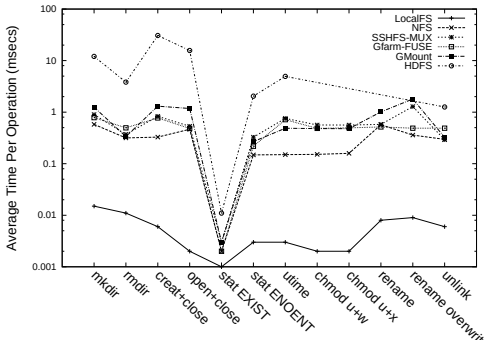
In **Fig. 12(b)**, we removed the results of "`stat EXIST`" to clarify the figure because the values are similar and small. The metadata servers of both Gfarm and HDFS are distant to their clients in WAN environments. Thus operation time increased because of network latency. In GMount, if client is looking up files that stored in affinity nodes, the operation

⋆1 The value $N : P : M$ reads that at most N concurrent unauthenticated connections are allowed to connect the sshd daemon with possibility P until maximum connection limit M is reached. See `man sshd_config(5)` for details.

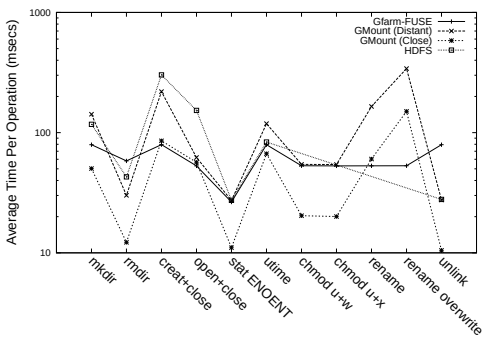
Table 1 Bandwidth (MB/sec) and RTT (ms) between sites in InTrigger

	hongo	chiba	okubo	mirai	kyushu	hiro	kobe	keio
hongo	-	592/6.2	70.8/3.8	77.9/24	479/27	32.1/25	227/19	94.7/3.8
chiba	787	-	44.2/6.3	65.2/29	495/31	301/29	28.1/29	94.9/9.2
okubo	47.5	49.3	-	45.0/28	54.8/29	54.8/27	52.1/28	80.7/9.7
mirai	46.5	45.3	12.2	-	46.5/49	48.4/48	29.0/41	28.1/28
kyushu	93.0	50.8	14.6	40.4	-	363/9.2	171/24	59.1/30
hiro	22.3	21.0	11.5	19.7	62.8	-	25.8/19	33.8/28
kobe	146	104	17.5	48.7	116	167	-	29.2/21
keio	91.0	60.3	20.8	11.1	26.3	15.4	48.1	-

can be quickly finished. However, for operations that need to lookup in all nodes, such as `create`, GMount also has to post messages to distant nodes.



(a) Metadata Operations in LAN



(b) Metadata Operations in WAN

Fig. 12 Metadata Operation Performance

Note that in GMount rename operations cost as triple time as in Gfarm. This is because rename in SFTP requires three round trips to finish, as mentioned in 5.1.1.

7. Conclusions

We proposed GMount, an ad-hoc distributed file system that can be built on-the-fly by non-privilege users. It is adaptive in the Grid environments because of its scalability, NAT/firewall transparency, and locality-aware operations. The usability of GMount is also demonstrated by its deployment and evaluation on the large-scale Grid platform.

Acknowledgements

This research is supported in part by the MEXT Grant-in-Aid for Scientific Research on Priority Areas project entitled "New IT Infrastructure for the Information-explosion Era".

References

1) Network file system. `http://www.nfsv4.org`.
2) Grid Data Farm. Online at `http://datafarm.apgrid.org`.
3) PVFS: Parallel Virtual File System. Online at `http://www.pvfs.org`.
4) Open Andrew file system. Online at `http://www.openfs.org`.
5) Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 9th ACM Symposium on Operating Systems Priciples (SOSP 2003)*, pages 29–43, New York, Oct 2003.
6) Kenjiro Taura. GXP: An interactive shell for the grid environment. In *Proceedings of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA 2004)*, pages 59–67, Charlotte, Apr 2004.
7) Miklos Szeredi. SSHFS: SSH Filesystem. `http://fuse.sourceforge.net/sshfs.html`.
8) Miklos Szeredi. FUSE: Filesystem in userspace. `http://fuse.sourceforge.net`.
9) OpenSSH. `http://www.openssh.org`.
10) UnionFS. `http://www.unionfs.org`.
11) Radek Podgorny. unionfs-fuse. Online at `http://podgorny.cz/moin/UnionFuse`.
12) IETF Secure Shell Working Group. SSH file transfer protocol. Online at `http://tools.ietf.org/wg/secsh/`.
13) Chris Rapier and Benjamin Bennett. High speed bulk data transfer using the SSH protocol. In *Proceedings of the 15th ACM Mardi Gras Conference*, pages 1–7, Baton Rouge, Jan 2008.
14) Hadoop. `http://hadoop.apache.org`.