# Orisync Usability Improvement

Philip Zhuang, Yinglei Wang
Oracle Corporation

## 1. MOTIVATION

File backup and distributed access is becoming a necessity for a lot of people. We often need the file systems on our our computers and mobile devices to stay in sync for easy access. Dropbox[1] is a popular solution for a lot of users. However, dropbox relies on centralized server to provide file synchronization. Centralization causes a lot of problems such as privacy and copywrite issues. We aim to develop a peer-to-peer file synchronization system. After some background research, we decide to build our system on top of Ori[2]. OriSync is a tool to enable automatic file system synchronization between different hosts. Its mains structure is shown in Figure 1. We have made several improvements to the original OriSync, and we also have some work in progress.

First, the original orisync only supports automatic synchronization between different machines. Local repository synchronization is important when there is no network connection between the two hosts which we want to synchronize. In this situation, we can enable automatic synchronization on mobile storage. Later, we can connect the mobile storage to the other host and do the same local sync. To make this possible, we implemented automatic synchronization between local repositories.

Second, the original orisync doesn't provide automatic snapshot. We need to manually take a snapshot after each change for the changed to be synced. Manually taking snapshots is obviously a trouble for users. We implemented automatic snapshot for the repositories registered in orisync to make orisync truly automatic.

Third, the original orisync is implemented as a daemon process. Other orisync commands only interacts with the configuration file on disk. This means, after we make a configuration change, we need to restart the orisync daemon process to read the new configuration. To eliminate the trouble of restarting the daemon process, we implemented Unix domain socket server for the orisync daemon process. Other orisync commands are implemented as Unix domain socket clients so that they can notify the daemon process of the changes.

Fourth, as a convenient tool, we implemented an orisync status command to show the status of the cluster. With this tool, the user can manage the cluster better.

Fifth, the original orisync can only be used by a single user on each host. Some hosts may have multiple users. We improved orisync so that it can be used by multiple users simultaneously on the same host.

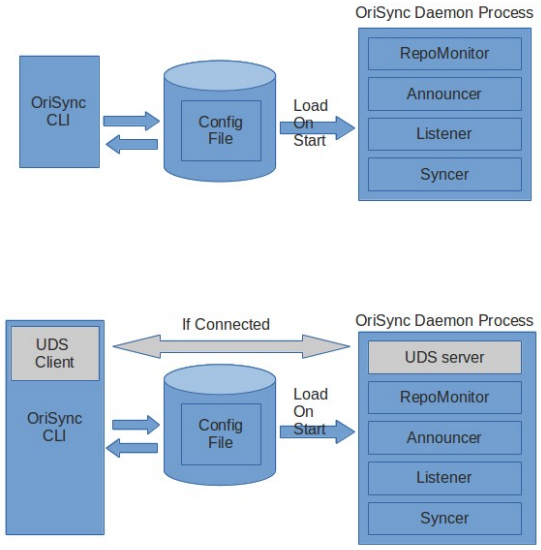


**Figure 1**

We also plan to add more features to orisync so that it fits the users' needs better, takes fewer storage and has better performance. These includes tracking the connection and usage pattern of the hosts so that we can have planned synchronization, policy based garbage collection, multi-cluster support and so on. We also fixed several bugs in the orisync code to make it work smoothly and reliably.

## 2. IMPLEMENTATIONS

### 2.1 Automatic synchronization of local repositories

The original OriSync only supports inter-host synchronization. For this purpose, it doesn't need all the replicas on each host. One replica from a replica group is enough to do the inter-host synchronization. Since all replicas in one replica group have the same UUID, the original orisync code uses a map for the registered repositories. The key is the repository UUID. To implement automatic synchronization of

local repositories, we need to loop through all the registered local repositories, and check whether it has a replica relationship with any other registered repositories. If so and the other repositories have some commits which the current repository doesn't have, we pull from the other repositories to get these commits. Merge or checkout is done after the pull. Therefore, we need the complete list of registered repositories. Considering the map structure is still needed for the inter-host synchronization, we added another list to record all the registered repositories on one host. The list is used for local repository synchronization. Now, any change in any replica of a replica group will propagate to all the replicas in the replica group, and propagate to all the hosts in the cluster. If one repository is in mobile storage, and the mobile storage is not present, orisync will throw a warning that this repository can't be opened and the other executions are unaffected.

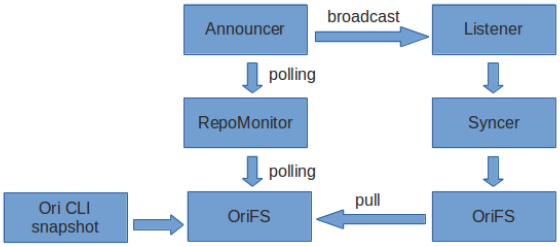


**Figure 2**

## 2.2 Communication of orisync CLI commands and orisync daemon process

The original OriSync CLI and daemon process structure is shown in Figure 2. The automatic synchronization threads are implemented in a stand-alone daemon process. When the daemon process is started, it loads the configuration information from a specific disk file. Other OriSync commands simply modify the on-disk configuration file and returns. In order to read the new configuration, the daemon process needs to be restarted. To implement the communication of orisync CLI commands and orisync daemon process, we added another thread in the daemon process. This thread will start a Unix domain server. Other orisync commands will create a Unix domain socket and exchange information with the server in the daemon process. We made use of the stream utilities provided by Ori to implement the exchange of information. Different commands are organized into an array of function pointers. This makes the coding very neat. One design consideration is that the orisync commands should also succeed when the daemon process is not running. In this situation, we simply log the configuration change to the configuration file. When the daemon process is started later, it can load the information from the disk file. If the daemon process is already running, the client will first log the information to disk file, and then notifies the daemon process of the change. Proper locking is used to ensure atomic operations and correct state.

## 2.3 Peer wared Orisync

Before our implementation, Orisync was unaware of its peer status. We decided to make Orisync keep track of its peer status for better observability and better repo management. In this way, users can keep track of all repos and hosts using orisync and identify problems like network issues. Also, by knowing its peer status, Orisync can control the pace of snapshotting to save system resources.

To make Orisync aware of its peers, we made some changes to both HostInfo class and RepoInfo class. We added "lasttime" attribute, "status" attribute and "down" attribute in HostInfo. "lasttime" shows the last time the machine received HostInfo from the host. "status" is a string message that shows the host status. And "down" is a boolean value that is true if the host goes down. Note that we can use HostInfo sent by Announcer periodically as heartbeat to detect if a host is down. If we detect current time - lasttime > HOST_TIMEOUT, we can assume the host is down and changed the status to "Down" with the last connect time and turn "down" to true. We created a watchdog thread in Orisync that wakes up every second to check if each host has connection time out. We also write other status to HostInfo when hosts are in other problems. For example, we will reject large time skew to prevent replay attacks. However, in many cases, users having two machines with large time difference will find orisync not operational. With this improvement, users can easily find out the problem when they see status "Time out of sync" using "orisync status" CLI, so they can sync time between two machines and get orisync working.

In RepoInfo, we added "mounted" and "peers" attributes. "Peers" attribute will keep track of all replicas of the repo in other hosts that are in sync with the repo. "Mounted" shows if the current repo is mounted on the filesystem and acts as a udsRepo. The "mounted" information will be broadcast to other server along with the entire HostInfo, so other hosts will know if the remote repo is mounted. This information is useful to our adaptive snapshot feature in the next section. Once we find a peer remote repo that is mounted, we will add the host to our peers set.

If the remote tells us the repo is no longer mounted or if we detect the repo host is down, we will remove the remote repo's host from the peers set.

We have also added a new CLI commands “orisync status” to show cluster status and improved "orisync list" to show all detail repo status. With "orisync status", we can easily tell what hosts are up and, what hosts are down or what hosts are in trouble. A sample CLI output is like:

```
$ orisync status
HOST        STATUS
host1       OK
host2       Down. Last connected Thu Dec  4 14:36:48 2014
host3       Time out of sync
```

With "orisync list", we can easily tell if the repo is mounted and list all its peers in sync. A sample CLI output is like:

```
$ orisync list
Repo                        Mounted Peers
/home/user/.ori/repo1.ori/  true    192.168.1.65
/home/user/.ori/repo2.ori/  false
```

## 2.4 Snapshot management improvement

We have improved the Orisync's snapshot management by making automatic and adaptive periodic snapshots.

Before we implemented periodic snapshot, orisync relies on users to take snapshots. Since orisync cannot sync the latest change without taking a snapshot, previous orisync cannot sync two repos continuously. In order to make continuous syncing work, we need to make snapshots automatically.

In our new RepoMonitor, we will do a snapshot on the repo before updating its RepoInfo. The snapshot is taken by sending a “snapshot” command on this repo to orifs through Unix domain socket. Note that if there is no change, no snapshot will be taken. Once there is any change, a new snapshot will be automatically taken orisync will announce the updated HostInfo immediately. On the other host, orisync’s Listener will catch the change in the updated HostInfo and notify the Syncer. The Syncer will pull and checkout the new snapshot. The entire latency from the snapshot taken to remote Syncer pulling the snapshot is only about 200ms, assuming negligible network latency. Note that if we change the orisync RepoMonitor interval into a very small value like 1 second, we can achieve almost instant update between two repos. In this way, we are able to continuously sync between two repos.

However, we don't want to try taking snapshots all the time if our host is standalone. For example, if you are working with your laptop offline, you may want to adjust your snapshot speed to a slower pace so that you don't end up with too many snapshots and consume too much CPU power. Because of this reason, we developed adaptive snapshot. Note that orisync is peer awared now (described in the previous section). We can use the peers information to decide whether we want fast or slow snapshot. If we have online and mounted peers, that means they are ready to sync continuously, and we can snapshot at a very fast speed like every second to keep file in sync. Otherwise, if we don't have peers, we can turn snapshot time to a much longer time like one minute just to save progress. We record lastSnapShot time in RepoInfo and check periodically by RepoMonitor to see if the repo is due for snapshot.

## 2.5 Improved sync latency

We have done several things to improve syncing latency. First, we merged Announcer into RepoMonitor so that as soon as RepoMonitor creates a new snapshot, it will call announce() to broadcast the change. Secondly, we create a queue called "Modified Repo Queue(MRQ)" and a condition variable mrqCV for Listener and Syncer communication. Once Listener received a HostInfo with a modified repo, it will put the modified repo along with the host it comes from to the MRQ and signals the Syncer. Syncer wakes up, dequeues the MRQ and check if it needs to pull. If a pull is necessary, it will pull directly with the given host and repo information. Note that after Syncer has done all the job, it will wait on the mrqCV again instead of sleeping for an interval so that it can handle the new pull as soon as possible. With these techniques, the sync latency between hosts is largely reduced. We will offer more performance results in the Evaluation section.

## 2.6 Garbage collection

As we enabled automatic snapshot, we will hit the problem as too many snapshots will take too much space. It's highly likely that we won't ever touch snapshots that are too old. So, we decide to do garbage collection on old snapshots. This feature is done both on orifs side and orisync side. Previously, orifs only kept track of user snapshots. Now, we also add a hashtable of orisync snapshots to localrepo and save it to file to keep it persistant across reboot. Note that we use timestamp as a key for orisync snapshots so that we can easily find or purge previous snapshots.

Garbage collection runs on the same watchdog thread that checks host status. Note that we don't expect garbage collection to run very often so it's OK to share the watchdog thread with host status checking. The watchdog will check repos periodically to see if they are due for garbage collection. If a repo is due for garbage collection (garbage collection time is defined by user, which is 10 hours by default), the watchdog will call orifs with a timestamp so that every snapshot before the timestamp will be purged. Orifs will then purge snapshots from old to new.

### 2.7 Multiple user support

Previous orisync doesn't support multiple users to run orisync on the same server. In order for multiple user support to work,we need to use SO_REUSEPORT to bind all orisync's listeners to the same port so that multiple Listeners can run at the same time. Note that our HostInfo is encrypted and we will drop requests from other clusters or the sender, so we don't need to worry about information exposed to other users or receive incorrect requests.

Also, previous orisync assumes all repos are under the same username. This prevents the same repo to be shared by different users or simply one user using different username on different machines. We fixed this problem by storing login username in HostInfo. We also added OriNet_Username() in liboriutil that uses getlogin() to retrieve username. With this fix, username will be broadcast along the repo, so the Syncer will know from which user it is going to pull the repo.

### 2.8 More robust locking and error handling

We have also created more robust locking and error handling for Orisync reliability. We have created a repo rwlock and a host rwlock per every repo and host. Since there are many race conditions going on with RepoMonitor, Listener, Syncer, UDSserver and Watchdog, the locks will prevent the race conditions. We also added more excpetion handlers and corner case checking. We have gone through a lot of tests to make sure Orisync is reliable.

### 2.9 Multi-cluster support

The baseline Orisync only supports one cluster. That is, a specific host is only allowed to be in one cluster. To add support for multi-cluster, we first need to change the Orisync command interface. Table 1 shows a list of commands which are changed.

| Description | Baseline | Multi-cluster |
|---|---|---|
| Add a repo | orisync add <repo> | orisync add <repo> <cluster> |
| Remove a repo | orisync remove <repo> | orisync remove <repo> <cluster> |
| Add a static host | orisync add <host> | orisync add <host> <cluster> |
| Remove a static host | orisync remove <host> | orisync remove <host> <cluster> |
| List all the registered repos | orisync list | orisync list <cluster> |

**Table 1 Orisync commands change to support multi-cluster**

For each host object, we added a map which maps the cluster name to the list of repos in the cluster. We also added a list of clusters that the host is in. The configuration of each cluster is stored in a file named "orisyncrc-<clusterName>". One special note is that all the configuration files on one host should have the same host ID. That means, when we initialize a cluster using "orisync init", we need to check whether the host has been assigned a host ID in another cluster. If so, we need to use the existing host ID. For the orisync daemon process, we need to make the following changes: 1. When the daemon process starts, it needs to load all the configuration files, and build a map which maps the cluster name to configuration objects. 2. For the RepoMonitor, we loop through all the clusters to monitor the repos in each cluster. 3. For the Announcer, we loop through all the clusters to broadcast its information. 4. For the Listener, when we receive a message, we need to check whether we are part of this cluster. If so, update related information. 5 For the Syncer, a repo only pulls from a remote repo if the two repos are in the same cluster.

## 3. Work in Progress

### 3.1 Selective snapshot pull

There is a good use case in Ori to use mobile device as a cache to sync between hosts in different locations. For example, your mobile device can sync between your work computer when you are at work. After work, your mobile device will carry the latest repo. As soon as you get home, your mobile device can sync with your home computer immediately. This utilizes fast LAN and can transfer large size of data in a small amount of time without using WAN.

However, the storage size on a mobile device is limited, and we don't want it to carry all the snapshots. We only want it to carry the snapshots up to a certain amount of time so that the other hosts can be updated with the recent snapshots. We would like to implement selective snapshot pull so that the mobile devices will only need to carry the "diffs" rather than keeping every snapshots.

# 4. EVALUATION

## 4.1 Correctness of local repository synchronization

We created one filesystem A with one text file. Then we made two replicas of A, here we call them B and C. All A, B and C are mounted and registered in OriSync. We randomly chose a filesystem and make some changes. After several seconds, we verified that the changes appeared in the other two filesystem.

To simulate mobile storage, we moved one filesystem B to another place on disk, effectively making B into B1. Before we remove B from OriSync, we got warnings saying that filesystem B can't be opened. However, the rest of the OriSync is functioning correctly. This shows the situation when the mobile storage is removed. We made another replica B2 from B1. When both B1 and B2 are registered, they began automatic syncing. This simulates the situation where the mobile storage is connected to another host. We plan to do real USB storage on two computers later.

## 4.2 Correctness of OriSync CLI and daemon process communication

Before we started the daemon process, we ran all the OriSync CLI commands. They are functioning exactly as before the change. We started the daemon process and ran the configuration change commands. The daemon process correctly got the change without restarting the process. For example, when a repository is added, this repository will start automatic synchronization. When a repository is removed, it stops automatic synchronization. The configuration file on disk also got the change.

## 4.3 Correctness of OriSync general syncing

We have created some test cases for Orisync general syncing in ori_tests/. The test case setup repos on two different hosts to sync automatically and verifies all Orisync functionalities and CLI work.

## 4.4 Correctness of multi-cluster support

We tested the multi-cluster support on three hosts A, B, and C (A is a laptop and B and C are virtual machines). On host A, we configured two clusters named "first" and "second". "first" contains repo A1 and A2 and "second" contains repo A3. On host B, we configured one cluster "first" which contains repo B1 and B2. On host C, we configured one cluster "second" which tains repo C1 and C3. Here, repos with the same number are replicas of each others (for example, A1, B1, and C1 are replicas of the same repo). Figure 3 shows the sharing relationship. We observed that B1 and B2 are automatically synced with A1 and A2, C3 is automatically synced with A3. This shows that repos in the same cluster are synced correctly. However, C1 is not synced with A1 and B1. This is expected because A and C are in the same cluster "second", but A didn't share A1 with C. B and C are not in the same cluster.

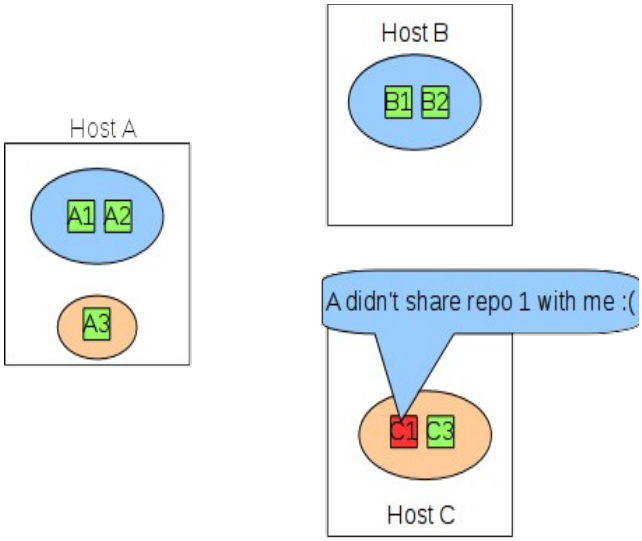


Figure 3. Multi-cluster example. There are three hosts: A, B and C, two clusters: "first" (blue circles) and "second" (orange circles). Repos with the same number are replicas. Green repos are successful synced and red ones are not.

## 4.5 Performance on Orisync

With automatic snapshot, we have run the following performance tests and the results are shown in Table 1 and 2.

In Table 1, we simply created a file by using "touch" and measured time taken on each step. 10 tests were run to take the average. In Table 2, we measured the time taken to sync a 2MB mp3 file. Note the syncing interval is 1 second. Both results show Orisync is able to sync files very quickly. The total latency from start snapshotting to pull done is only about 300-400ms.

**Table 2. Performance statistics on Orisync with file creation**

| | Avg time since file creation | Avg time between each step |
|---|---|---|
| File created | 0ms | 0ms |
| Start snapshot | 417.25ms | 417.25ms |
| Snapshot done | 527.34ms | 108.09ms |
| Start announce | 527.47ms | 0.12ms |
| Announce done | 527.57ms | 0.10ms |
| Start pull | 699.57ms | 172.00ms |
| Pull done | 792.31ms | 92.74ms |

**Table 3. Performance statistics on Orisync with 2MB mp3 file synchronization**

| | Avg time between each step (ms) |
|---|---|
| Start snapshot | 0 |
| Snapshot done | 230.179 |
| Start announce | 0.136 |
| Announce done | 0.098 |
| Start pull | 0.413 |
| Pull done | 93.478 |

## 5. CONCLUSION

Our implementation has largely increased the usability of orisync. We have made ori sync repositories continuously, enabled ori to sync local repositories and enabled multiusers to use orisync on a same server. We have also improved orisync CLI that exposed more information to user and enabled user to dynamically configure orisync without restarting it.

We believed these features will make orisync more usable and make it appeal to more people. We'll continue to work on some other useful features and hopefully these will make Ori more and more popular.


## 6. ACKNOWLEDGMENTS

Firstly, thanks to Prof David Mazières for referring us to Ori project when we told him similar project ideas to Ori.

Also, many thanks to Ali Mashtizadeh. He offered us a lot of ideas and suggestions on Orisync improvement and helped us understand the existing Ori project.


## 7. GIT Repositories

https://bitbucket.org/orifs/ori-orisyncng

## 8. REFERENCES


[1] Dropbox Inc. http: //www.dropbox.com/, December 2014.

[2] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frang Huang, David Mazières. Replication, History, and Grafting in the Ori File System. In *Proceedings of the 24th Symposium on Operating Systems Principles*, November 2013.