Vol.2011-HPC-130 No.39
2011/7/28

Cynk: A Hybrid Rsync and SSH Filesystem for Cloud Computing

Nan Dun,[†1] Sugianto Angkasa,[†1] Kenjiro Taura[†1]
and Ting Chen[†1]

Cynk is a hybrid file system using rsync and SSH for data-intensive cloud computing. By automatically synchronizing the local file system with a cloud storage, Cynk enables users to transparently access local/remote data when they are online and continue working when disconnected from network. The hybrid architecture of Cynk means that it can allow users to simutaneously access locally synchronized/cached data or online remote data over the network via a uniform file system interface. Cynk uses the rsync tool with a partially reasoning based protocol to synchronize files from local to remote file systems and vice versa. It only requires the installation of client on local side. By seamlessly bridging the local file system and cloud storage, Cynk especially simplifies the work cycle of developing, testing, and deploying data-intensive applications.

1. Introduction

Cloud platform services, or Platform as a Service (PaaS), have provided scalable and elastic computation and storage resources for various applications[1),6),9),12)]. Data-intensive scientific applications[7)], usually requiring significant computation power to process a large amount of data, are an important applications category that can particularly benefit from the cloud platform services.

Though most of cloud services already have a friendly user interface to allow users seamlessly scale the execution of applications to multiple computer servers on demand, there are still some inconveniences that prevent the routine practice from being as easy as we expected.

First, users usually develop and test their applications in their own desktop machines for convenience in early stage, and then explicitly copy the program and data to cloud for further test or real execution. This is *efficient* and *economic* for users because: i) directly operating in cloud usually suffers from a much higher latency than working on local desktop machines; ii) developing and testing are generally time-costing task but require few computation and storage, while conducting this task on cloud will receive an additional charge based on the time or storage usage of the cloud server instances[1)].

Second, users prefer to separate the data storage locations for different datasets used in testing and real application execution. Current personal computers are usually equipped with hundreds of GBs to several TBs of disks, but they are still not enough to store hundreds of TBs or PBs of data that are typically processed by large applications. Additionally, users sometimes only need to access a small fraction of the entire dataset for testing *without* copying the entire dataset to local storage, which suggests that the large data in remote cloud storage should be easily accessible over the network from user's local desktop.

Being aware of these problems when we conduct our routine practice, we design and implement Cynk, a hybrid filesystem using rsync and SSH for data-intensive cloud computing, aiming to simplify the work cycle of developing, testing, and deploying data-intensive applications. Cynk can automatically synchronizing the local file system with a cloud storage, which enables users to transparently access local/remote data when they are online and continue to work when they are disconnected from network. The *hybrid* architecture of Cynk means that it can also allow users to seamlessly and simultaneously access locally synchronized/cached data or online remote data (without copying) over the network via a uniform file system interface. Cynk especially targets the typical usage scenario in practice as follows:

- Users develop and test their component programs and workflows in their desktop machine with *small datasets*, which can be done *offline*, e. g., in a commuter train or during a trip.
- Once the local development is done, users are able to immediately process *large datasets* with remote resources when they are *online*, without the need to recompile the application components and explicitly copy/move data around.
- While the component programs and small dataset are offline available for

†1 The University of Tokyo

testing purpose, user are still able to test it with large datasets by seamlessly access it when the remote storage is online without copying anything.

Therefore, Cynk allows users to effortlessly bridge the local desktop and a remote storage system that is co-located with computation resources. The storage system can be a commercial cloud storage, such as Amazon Simple Storage Service[1], or a distributed file system that federates multiple servers for data sharing service in the dedicated environments, such as Lustre[11] for clusters, Gfarm[16] or GMount file system[3] for distributed clusters.

2. Hybrid Remote File System

Cynk is a user-level file system, which can be installed by non-privileged users and *only* needs to be installed on the local side. We assume that an Secure Shell (SSH) server[17] is always available on the remote side.

Figure 1 illustrates the *hybrid namespace* in local filesystem created by mounting a remote filesystem using Cynk. The local namespace is rooted at the Cynk mountpoint `working_directory` and its target is a remote directory tree rooted at `workflow`. Cynk creates a virtual namespace that is identical to the target remote namespace. By default, all directories and files under the `workflow` directory are kept synchronized with `working_directory`, i. e., `source`, `binary`, `small_dat` directories, which are illustrated by real lines in Figure 1. Directory is also able to be specified as an *online directory*, which is *not* synchronized but accessed directly over the network, such as `big_dat` directory.

By this design, users can: i) transparently access local/remote data when they are online, ii) continue working offline when they are disconnected from network, and iii) separate the data for testing and full execution. As a result, the application can be developed completely in the local file system with full datasets access, and then users can just execute the application with full datasets and computation resources.

2.1 Offline Data Synchronization

The offline data requires an initial replication from one side to another side, and further synchronizing operations when files are updated on either side. In Cynk, we use rsync[19] to perform the file synchronization. Rsync is a popular software utility for synchronizing files/directories between two file systems, using

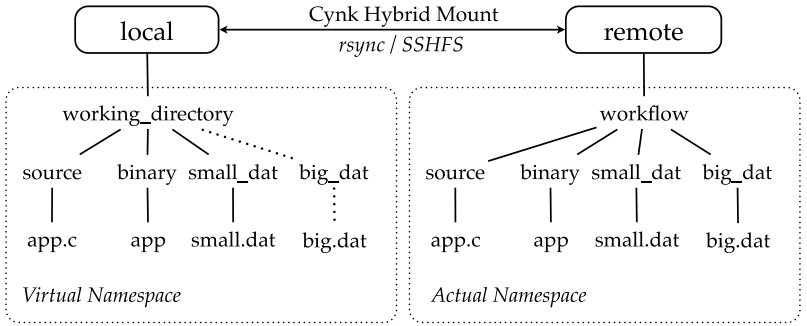

Fig. 1 Hybrid namespace created by Cynk mount

an algorithm to minimize the data transfer by delta encoding[18]. One important feature of rsync is that the synchronization can be performed by only one side, suggesting that there is no need to install additional software on the other side. Rsync can use its native protocol (requiring installation of rsync on both sides) or via a remote shell such as Remote Shell (RSH) or SSH, to conduct file operations.

We present the details of synchronization protocols in Section 3.

2.2 Online Data Access

The online access for remote data only requires a proper file transfer protocol for file manipulation via the file system interface. To achieve this, we use the same approach of an existing remote filesystem called SSH Filesystem (SSHFS)[15]. SSHFS is a file system based on Filesystem in Userspace (FUSE) and SSH File Transfer Protocol (SFTP)[17]. It enables users to mount the file system of a remote machine on local file system via usual SSH channel. Since SSH is widely available in most Unix servers, thus users do not have to run additional daemons in advance and the data transfer is as secure as SSH.

3. Synchronization Protocol

Since Cynk is designed to be installed *only on user's desktop side*, we are only able to capture the file system events (i. e., via FUSE interfaces) invoked by user's manipulations on local files. Besides, the file updates can also happen on the remote server side, leading an inconsistency of files on two sides. Therefore,

the challenge is that we *lack the knowledge of file updates of the remote side* to conduct a reasoning of the temporal order of update events from both sides.

Our goal is to design a synchronization protocol based on existing rsync operations to achieve the functionality we proposed, even without a strict conformity of the file consistency reasoning assurance. Or, more importantly, the file consistency can be guaranteed under some specific circumstances that are common usage cases in practice.

3.1 Semantics

We use the following notations to describe the updated file and the status of file systems before and after the synchronization. S denotes the status of file systems right after the *last* synchronization, thus the files are identical for both local and remote file systems. Arrow "$\rightarrow$" denotes a transition with any updates/changes between two synchronizations. L and R are the statuses of local and remote file systems, respectively, right before the *next* synchronization. f denotes the name of a file that may be updated (i. e., created, written, or removed) during the period between two synchronization.

Depending on the file system statuses of L and R, there are eight cases that need to be considered in total:

Case 1: $f \in S \rightarrow f \in L$ and $f \in R$. The file existed at the last synchronization point and it still exists in both sides. It may have been modified, or deleted and then created since the last sync point. But any way, it does exist right before the synchronization. Thus, we leave whichever is newer (based on the modification time of files) between f in L and f in R, i. e., transferring the newer file from its original side to the other side.

Case 2: $f \notin S \rightarrow f \in L$ and $f \in R$. The file did not exist at the last synchronization point but appears on both sides. Somehow, both L and R must have created f. Clearly there is conflicting, but we have no option other than leaving whichever is newer between f in L and f in R, i. e., transferring the newer file from its original side to the other side.

Case 3: $f \in S \rightarrow f \in L$ and $f \notin R$. The file must have been deleted in R at least once. There may be conflicting cases, such as it was deleted locally and then it was recreated again. But in any case, there was a "delete" operation on R side and we have no way to tell that there is a local create operation after the remote delete. Thus in this case, deleting f in L is the only reasonable way.

Case 4: $f \notin S \rightarrow f \in L$ and $f \notin R$. The L side must have created f at least once. There may be other conflicting operations (e. g., R created f and then deleted it). But there is no way to tell that delete operation occurred after the last creation of f on L. Thus, we leave f as it is, i. e., transferring f from L to R.

Case 5: $f \in S \rightarrow f \notin L$ and $f \in R$. The opposite of case 3. Deleting f in R is the only reasonable way.

Case 6: $f \notin S \rightarrow f \notin L$ and $f \in R$. The opposite of case 4. Transfer f from R to L is the only reasonable way.

Case 7: $f \in S \rightarrow f \notin L$ and $f \notin R$. Somehow, both sides must have deleted f at least once. Clearly there is conflicting, but we have no option other than leaving it alone. There is nothing to do for this case.

Case 8: $f \notin S \rightarrow f \notin L$ and $f \notin R$. Both side can have created and finally deleted f. Clearly conflicting, but we have no option other than leaving it alone. There is nothing to do for this case.

3.2 Rsync Primitives

The synchronizing operation performed by rsync can be well controlled by specifying its command options (see manual page of rsync for details[19]). Here, we focus on discussing how to control fine-grained rsync operations by using the combination of different options in the context of synchronization semantics presented in Section 3.1. We use "$\Rightarrow$" to denote a rsync command to be executed.

3.2.1 Basic Operations

There are several important command options used to control the rsync behaviors. We list them as a reference for further description.

Default action: $\Rightarrow$ `rsync src dst`

- Files existing only on `src` will be transferred to `dst`.
- Files existing only on `dst` will be ignored.
- Overwrite files with the same name but *different* size or modification time, transferring from `src` to `dst`.

Update action: $\Rightarrow$ `rsync src dst -u`

- Same as default action, but does not update/overwrite files that are newer

on `dst`.

Delete action: ⇒ `rsync src dst --delete`

- Same as default action, but delete files that exist *only* on `dst`.

Exclude action: ⇒ `rsync src dst --exclude-from=list`

- Same as default action, but do nothing on files that are listed in `list`.

3.2.2 Two-way Synchronization

Now we discuss the effect of basic operations on the eight cases described in Section 3.1, with the consideration of file updates on both sides for two-way synchronization. Basically, a two-way synchronization consists of two directions of transmissions: a *pull phase* that transfers files from R to L, and a *push phase* that transfers files from L to R.

Read only: There are only read operations on both L and R since S. Thus following two-way synchronization command does not change anything since files of both sides are identical.
⇒ `rsync R L && rsync L R`

Write: There are write operations on either/both L or/and R since S. According to case 1, we keep files updated to the latest version of copies.
⇒ `rsync R L -u && rsync L R -u`

Create: There are create operations, but *no delete operations*, on either/both L or/and R since S. According to case 2, 4, and 6, we keep files newly created and updated to the latest version of copies.
⇒ `rsync R L -u && rsync L R -u`

Delete: There are delete operations, but *no create operations*, on either/both L or/and R since S. According to case 3, 5, and 7, we delete the files on both sides.
⇒ `rsync R L -u --delete && rsync L R -u --delete`

From above analysis, it is not hard to find that, when using "-u" and "–delete" options together, there can be conflicts if file updates include both create and delete operations. Figure 2 and Figure 3 illustrate the conflicting cases.

A straightforward solution for these problems is to use "–exclude-from" options. Specifically, a file is logged into a list when it is created or deleted, since we are able to capture the file system events of L by FUSE. Then before the pull phase of synchronization, the files in this list are all excluded from being touched

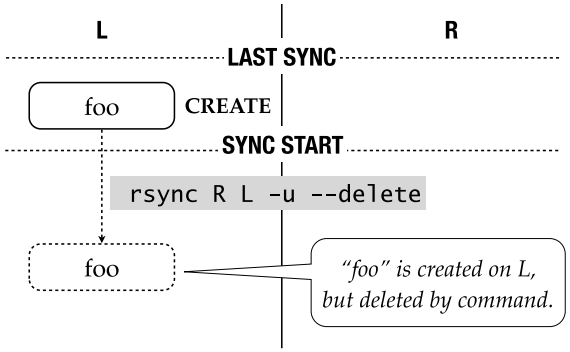

Fig. 2 Conflict when creation on L

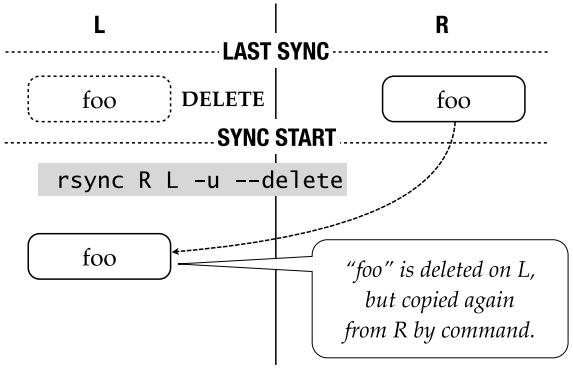

Fig. 3 Conflict when deletion on L

by rsync operations. For example, in Figure 3 and Figure 2, the command will equivalently becomes as "`rsync R L -u --delete --exclude=foo`". As a result, a file created in L will not be deleted by pull command (see Figure 2). A file deleted in L will not be copied back from R by pull command (see Figure 3), and a following push command "`rsync L R -u --delete`" will delete the file in R, resulting a compliance with the semantics in Section 3.1.

We use a *born list* to record the files with the create type events, and a *dead list* to record the files with the delete type events. Note that a create type event is

not only consists of *creation*, but also includes other events such as `mknod`, `mkdir`, `rename` (to), and `link` (to). Similarly, a delete event can be `unlink`, `rmdir`, and `rename` (from).

Since a file can be created, deleted, created again, and deleted again, etc., the born list and dead list should be updated accordingly upon create/delete events. The algorithms for lists updating is straightforward and simple:

- *On create event:* Check if the file is already logged in the dead list. If exist, remove it from dead list and return. Otherwise, insert the file into born list.
- *On delete event:* Check if the file is already logged in born list. If exist, remove it from born list and return. Otherwise, insert the file into dead list.

As a result, the complete two-way synchronization protocol is:

- Logging the create and delete types events respectively, while maintaining a born list and a dead list.
- Synthesize an exclude list by adding all files in born and dead lists, and use this list to conduct a pull phase by
  ⇒ `rsync R L -u --delete --exclude-from=exclude_list`
- Synthesize another exclude list by addiing files in dead list only, and use this list to conduct a push phase by
  ⇒ `rsync L R -u --delete --exclude-from=exclude_list`

4. Implementation

Using FUSE framework, Cynk implements a user-level filesystem by integrating rsync with the synchronization protocol addressed in Section 3 and SSHFS (see Figure 4).

FUSE[14], consisting of a kernel module and a userspace library, is a popular framework that allows users to develop their own file systems without touching the kernel. By implementing FUSE interfaces, users can map file system calls to specific userspace application and thus existing binaries can run on top of the local file system without modification.

There a several reasons that we use FUSE rather than other approaches to implement Cynk. First, FUSE can guarantee the fully capturing of file system events. While in other file system event monitoring utility such as inotify[8], its implementation mechanism limits its monitoring scalability of large amount

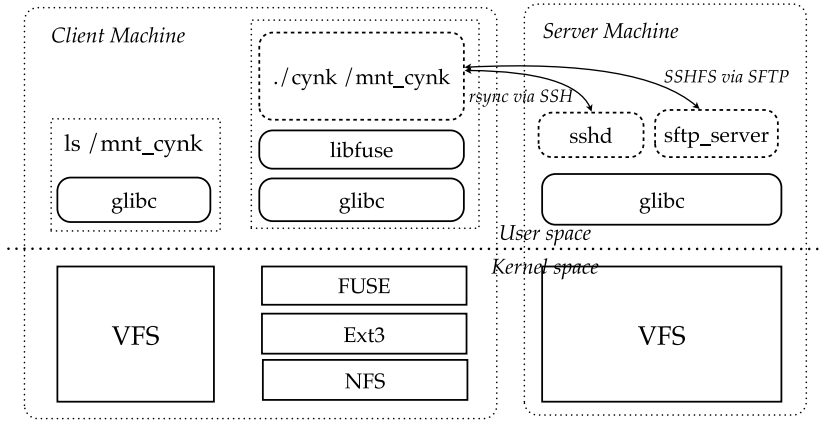

Fig. 4 Implementation architecture of Cynk

events. For example, there are race conditions for recursively directory watching which can cause events to be missed if they occur in a directory immediately after that directory is created. Second, FUSE allows us to perform the file system blocking to avoid the race condition if applications continue to access the file system while synchronization is in progress. Third, FUSE framework allows us to integrate SSHFS in a more straightforward approach.

Cynk is implemented in about 5,000 lines of C code, of which 4,000 lines are adopted from SSHFS source code[15] and another 1,000 lines are dedicated to implementing the two-way synchronization protocol.

5. Evaluation

Since Cynk is only used for file synchronization, we measure its synchronization performance and use Dropbox[2] as a reference. Our experimental environment consists of two notebook machines with Linux installed. We install Dropbox on both machines which refer to the same account, but we only run Cynk on one of machines, which is denoted as local machines. Then, we use following dataset to test Cynk and Dropbox with the same configurations.

- *Data 1:* 1000 small size (600 B - 1.2 KB) text files
- *Data 2:* 2000 small size (600 B - 1.2 KB) text files

- *Data 3:* Several medium size (10[MB] - 50 MB) files consisting of video files, binary files, and compressed files, with a total size of 112.3 MB
- *Data 4:* Several big size (50 MB - 150 MB) files consisting of video files, binary files, text files and compressed files, with a total size of 1.9 GB.
- *Data 5:* Several big size (150 MB - 350 MB) files consisting of video files, binary files, and compressed files, with a total size of 1.9 GB.

Table 1 List of Cynk and Dropbox synchronization time. Note that the number cannot be compared directly because Dropbox always needs to communicate with its file servers.

Operation	Cynk (sec)	Dropbox (sec)
Move data 1 via WiFi connections	59.8	171.2
Delete data 1 via WiFi connections	75.2	61.4
Move data 2 via WiFi connections	69.2	314.6
Delete data via WiFi connections 2	90.6	64.8
Move data 3 via WiFi connections	408	941
Delete data 3 via WiFi connections	79	295
Move data 4 via LAN connections	226	459
Update data 4 via LAN connections	118	1413
Delete data 4 via LAN connections	27	13
Move data 5 via LAN connections	227	1347
Update data 5 via LAN connections	96	2794
Delete data 5 via LAN connections	58	8

Table 1 shows the list of synchronization time by using Cynk and Dropbox. Note that we are unable to compare the time directly because Dropbox uses a centralize storage to store and distribute files to multiple clients, where the latency between client and file servers are much higher than the latency between two machines used by Cynk.

First, Cynk is better in term of the ease of usage. When using Dropbox, user needs to install and configure Dropbox on every clients. While for Cynk, user only need to run Cynk from local site and it is able to synchronize with multiple remote machines specified with just one command for each remote site. Besides, Dropbox only synchronizes the files in Dropbox folder, which means that user needs to explicitly move/import files to the Dropbox folder. Cynk doesn't have this restriction because it is able to synchronize with any folder specified by the command arguments.

Second, Cynk is better in case of moving updated files. In the experiment, we choose several files of data 4, 5, and 6, and make some changes with the portion of data with a size of 10 MB - 50 MB. It is ideal if both tools only move these updated 10 MB - 50 MB data. For text files both Cynk and Dropbox perform well because they only moved the changed part of files. But for binary, video, and compressed files, Dropbox is unable to identify the changes and then move the entire changed files. However, Cynk only moves the changed portion of data. This is because Dropbox uses a text-based versioning system to differentiate updated files and thus unable to detect the changes for non-text files. On the other side, Cynk, which is actually using rsync, uses the delta differencing approach to identify the difference of updated file and transfer that part of data only.

However, Dropbox is better if users would like to synchronize files with other multiple machines. For Dropbox, the time for distributing data to multiple machines remains almost the same no matter the number of remote sites. But for Cynk, the time required for distributing data to multiple machines will rise.

6. Related Work

Existing file synchronization utilities for cloud storage include Dropbox[2], Syncplicity[13], Ubuntu One[20], and Windows Live SkyDrive[10]. Dropbox is a client-based application that enable users to share and synchronize their personal files across multiple platforms. Syncplicity is also a desktop-based file backup and synchronization service, which allows other popular online service such as Google Docs[5], Zoho[21], and Facebook[4]. Ubuntu One[20] is a Ubuntu-dedicated client application for file synchronization across several platforms. Windows Live SkyDrive[10] is a web-based file hosting server on cloud storage with deep integration with various Windows Live online services.

However, Cynk is particularly different from above file hosting/synchronization services in following aspects. First, Cynk targets to help users reduce the developing effort of running data-intensive applications on cloud service, while all above file hosting services focus on providing an easy personal data sharing/distributing services across different platforms and devices. Second, Cynk allows users to *directly* synchronize their files with personal servers, dedicated clusters, or commercial cloud storage. On the other side, existing

file services are supported file servers that are managed in a centralized manner.

7. Conclusions and Future Work

We have developed Cynk, a file system that simplifies the practice of developing and executing data-intensive applications on cloud platforms. Cynk achieves its usability and simplicity by 1) integrating rsync and SSHFS into one hybrid remote file system that enable transparent access of online and offline data in remote storage system from user's desktop, and 2) a two-way synchronization protocol based on partial file updates knowledge.

Our future work includes developing an additional synchronization protocol that is strictly compliant with the temporal order of file updates on both local and remote sides. For example, we can implement a Cynk daemon for remote side if applicable, so the complete file system events can be acquired for conflicts reasoning. Another direction of enhancement is to make Cynk *smarter* by designing a better synchronization triggering mechanism. For example, Cynk will trigger the synchronization by detecting the specific patterns of file system activities, instead of periodic synchronizing in current implementation.

Cynk is an open source software and online available at `http://cynk.googlecode.com/`.

References

1) Amazon.com: Amazon Web Services, Amazon.com, Inc. (online), available from ⟨http://aws.amazon.com/⟩ (accessed 2011-06-23).
2) Dropbox: Dropbox, Dropbox Inc. (online), available from ⟨http://www.dropbox.com/⟩ (accessed 2011-05-05).
3) Dun, N., Taura, K. and Yonezawa, A.: GMount: An Ad Hoc and Locality-Aware Distributed File System by Using SSH and FUSE, *Proceedings of the 9th IEEE International Symposium on Cluster Computing and the Grid*, CCGrid '09, pp. 188–195 (2009).
4) Facebook, Inc.: Facebook, (online), available from ⟨http://www.facebook.com/⟩ (accessed 2011-06-23).
5) Google, Inc.: Google Docs, (online), available from ⟨http://docs.google.com⟩ (accessed 2011-06-23).
6) Google.com: Google App Engine, Google, Inc. (online), available from ⟨http://code.google.com/appengine/⟩ (accessed 2011-06-23).
7) He, T., Tansley, S. and Tolle, K.(eds.): *The Fourth Paradigm: Data-Intensive Scientific Discovery*, Microsoft Research (2009).
8) Linux man page: inotify, (online), available from ⟨http://linux.die.net/man/7/inotify⟩ (accessed 2011-06-23).
9) Microsoft: Windows Azure, Microsoft Corporation (online), available from ⟨http://www.microsoft.com/windowsazure/⟩ (accessed 2011-06-23).
10) Microsoft Corporation: Windows Live SkyDrive, (online), available from ⟨http://skydrive.live.com/⟩ (accessed 2011-06-23).
11) Oak Ridge National Laboratory: Peta-Scale I/O with the Lustre File System, Technical report (2008).
12) Salesforce.com: Salesforce, Salesforce.com (online), available from ⟨http://www.salesforce.com/⟩ (accessed 2011-06-23).
13) Syncplicity, Inc.: Syncplicity, (online), available from ⟨http://www.syncplicity.com/⟩ (accessed 2011-06-23).
14) Szeredi, M.: FUSE: Filesystem in Userspace, (online), available from ⟨http://fuse.sf.net/⟩ (accessed 2011-05-08).
15) Szeredi, M.: SSH Filesystem, (online), available from ⟨http://fuse.sf.net/sshfs.html⟩ (accessed 2011-05-08).
16) Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol.28, pp.257–275 (2010).
17) The OpenBSD Project: OpenSSH, (online), available from ⟨http://www.openssh.org⟩ (accessed 2011-06-23).
18) Tridgell, A.: Efficient Algorithms for Sorting and Synchronization, PhD Thesis, Australian National University, Canberra, Australia (1999).
19) Tridgell, A., Mackerras, P. and Davison, W.: Rsync, (online), available from ⟨http://rsync.samba.org/⟩ (accessed 2011-05-05).
20) Ubuntu: Ubuntu One, (online), available from ⟨http://one.ubuntu.com/⟩ (accessed 2011-06-23).
21) Zoho, Inc.: Zoho, (online), available from ⟨http://www.zoho.com⟩ (accessed 2011-06-23).