TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 08/11 pp286-297
Volume 18, Number 3, June 2013

Coarse-Grained Cloud Synchronization Mechanism Design May Lead to Severe Traffic Overuse

Zhenhua Li, Zhi-Li Zhang, and Yafei Dai*

Abstract: In recent years, cloud sync(hronization) services such as GoogleDrive and Dropbox have provided Internet users with convenient and reliable data storing/sharing functionality. The *cloud synchronization mechanism* (in particular, how to deliver the user-side data updates to the cloud) plays a critical role in cloud sync services because it greatly affects the *cloud operation cost* (in terms of sync traffic) and *user experience* (in terms of sync delay). By comprehensively measuring tens of popular cloud sync services, we find that their cloud sync mechanisms differ greatly in sync performance and design granularity. Quite surprisingly, some very popular services (like GoogleDrive and 115 SyncDisk) utilize a quite coarse-grained cloud sync mechanism that may lead to severe traffic overuse. For example, updating 1-MB data may sometimes result in 260-MB sync traffic. In this paper, we conduct a comparative study of various cloud sync mechanisms by analyzing their respective pros and cons under different situations, unravel the pathological processes for their traffic overuse problems, and finally provide insights/solutions for better choosing/designing a cloud sync service.

Key words: cloud storage; data synchronization; operation cost; user experience; design granularity

1 Introduction

In recent years, cloud synchronization services, such as GoogleDrive, Dropbox, 115 SyncDisk[1], SugarSync, Amazon CloudDrive, SkyDrive, and other services[2-13], have provided Internet users with convenient and reliable data storing/sharing functionality. Dropbox is reported to possess more than 100 million users[14] and 115 SyncDisk (also called the "China's Dropbox") owns over 30 million users[15]. GoogleDrive was released in 2012 and then quickly obtained a large user group through both Google's advertisements and its seamless integration with the popular GoogleDocs[16] service. Most cloud sync services require or encourage their users to install a designated client (software) and to assign a designated local folder (named "sync folder"). The user can add a file into or directly modify a file in this sync folder that is then automatically synchronized with the cloud by the client.

Since most cloud sync services limit the per-user quota of cloud storage space (e.g., a Dropbox user usually owns around 2-GB free storage quota and a GoogleDrive user owns 5-GB free storage quota), their users would often modify existing files rather than backup new files in their sync folders. Therefore, the *cloud synchronization mechanism* (in particular, how to deliver the user-side *data updates* to the cloud) plays a critical role in cloud sync services because it greatly affects the *cloud operation cost* (in terms of sync traffic) and *user experience* (in terms of sync delay). Specifically, on one hand, the data sync traffic makes up a significant portion of the cloud operation cost[17], so we hope the consumed network traffic for synchronizing a data update can be as little as possible. On the other hand, once a data update happens, we hope the sync delay can be as short as possible (namely,

• Zhenhua Li and Yafei Dai are with the Department of Computer Science and Technology, Peking University, Beijing 100871, China. E-mail: {lzh, dyf}@net.pku.edu.cn.
• Zhi-Li Zhang is with the Department of Computer Science and Engineering, University of Minnesota — Twin Cities, Minneapolis, MN 55455, USA. E-mail: zhzhang@cs.umn.edu.
∗ To whom correspondence should be addressed.
Manuscript received: 2013-02-25; accepted: 2013-03-01

a novel data update can be synchronized to the cloud as soon as possible) to enhance the user experience[1]. Furthermore, in practice we hope the *sync process* can be as simple as possible, so that the client software is easy to develop and the cloud platform is easy to deploy. Unfortunately, one "hope" is often conflicting with another (under specific application situations), as demonstrated by the following typical mechanisms:

(1) *Update-triggered Full-file Sync (UFS)* in 115 SyncDisk: Once a data update happens, the full content of the updated file f is delivered to the cloud. Thereby, the sync delay is minimized to several seconds when f is small and the sync process is almost as simple as directly uploading a file to the cloud (like using the FTP protocol). However, the resulting sync traffic is proportional to the size of f rather than the size of the data update, so the sync delay is usually quite long when f is large. Still worse, in the case of *incremental data updates* (namely, when a file is incrementally updated rather than updated in a single pass), UFS may deliver numerous full files to the cloud.

(2) *Timer-triggered Full-file Sync (TFS)* in GoogleDrive: Once a data update happens, a timer is set to watch whether there will be following data updates in the subsequent T seconds (e.g., $T \approx 4.2$ s for GoogleDrive). If yes, this data update will be ignored; otherwise, the full updated file is delivered to the cloud. Compared with UFS, TFS can *sometimes* reduce the sync traffic but obviously prolong the sync delay.

(3) *Update-triggered Delta Sync (UDS)* in Dropbox: Once a data update happens, only the latest data update(s) (i.e., the "delta" file, Δf, between the updated file f and the corresponding latest cloud-stored file f') are delivered to the cloud. Compared with UFS, UDS can significantly reduce the sync traffic and shorten the sync delay when f is large but Δf is small. Nevertheless, delta sync is more complicated than full-file sync, since the delta sync process involves at least three steps[18]: (a) the client retrieves the metadata of f' from the cloud; (b) the client computes the "delta"[19] (or says "binary diff"[20]) file between f and f'; and (c) the client delivers the "delta" file to the cloud.

(4) *Timer-triggered Delta Sync (TDS)* in SugarSync: TDS is similar to TFS except that only data updates are delivered to the cloud, so TDS can also significantly reduce the sync traffic and shorten the sync delay when the updated file is large but the "delta" file is small. Nevertheless, TDS is more complicated than both TFS and UDS.

(5) *Manual or Periodical Sync (MPS)* in Amazon CloudDrive: The updated file or data updates are delivered to the cloud only when the cloud sync process is manually or periodically performed (the period is usually configured as long as one hour, one day, and so forth). In general, if equipped with delta sync, MPS should be the most traffic-saving way to use a cloud sync service, but the sync delay becomes extremely long and it is inconvenient for data sharing among multiple users.

By comprehensively measuring tens of popular cloud sync services as listed in Table 1, we find that their cloud sync mechanisms differ greatly in sync performance and design granularity. In particular, quite surprisingly, some very popular services (like GoogleDrive and 115 SyncDisk) utilize a quite *coarse-grained* cloud sync mechanism that may lead to *severe* traffic overuse. Nowadays, users are employing cloud storage services to do more and more "complicated" things, such as collaborative editing[16,21], database storage[22,23], and even media streaming[24,25], that involve *frequent incremental data updates*. As a simple example, if we append 5 K random[2] bytes to an existing empty file in the GoogleDrive sync folder per 5 s until the total appended bytes reach 1 MB, the resulting sync traffic will be around 260 MB[3]. On handling the same data update pattern, the resulting sync traffic is around 186 MB for 115 SyncDisk. However, neither GoogleDrive nor 115 SyncDisk has ever told its users

[1] Generally speaking, the user is aware of (or sensitive to) the sync delay because he can notice it from the client status or notification, but unaware of the sync traffic because no cloud sync client tells the user about the traffic.

[2] To our knowledge, most cloud sync clients compress the updated file or data updates before delivering it/them to the cloud, which makes it quite inconvenient to figure out the real size of the compressed file or data updates. To facilitate our computation, the appended bytes are *randomly* generated so they can hardly be compressed by the client before delivered to the cloud.

[3] Our Internet access bandwidth is 4 Mbps and the experiment is performed by using the GoogleDrive client (version 1.2.3123.0250) in Minneapolis, MN, US in 2012. Besides, if we use an existing non-empty file rather than an empty file, the resulting sync traffic will be more than 260 MB.

Table 1 Classification of popular cloud sync services in terms of their cloud sync mechanisms.

Cloud sync mechanism	Cloud sync services
UFS: Update-triggered Full-file Sync	115 SyncDisk[1], UbuntuOne, Kanbox[4], Kuaipan[5], ⋯
TFS: Timer-triggered Full-file Sync	GoogleDrive, SkyDrive, Baidu CloudDisk[3], ⋯
UDS: Update-triggered Delta Sync	Dropbox, 360 CloudDisk[8], IDriveSync[7], Everbox[9], ⋯
TDS: Timer-triggered Delta Sync	SugarSync, ⋯
MPS: Manual or Periodical Sync	Amazon CloudDrive, Box[2], SpiderOak, CrashPlan, ⋯

to avoid generating such (frequent incremental) data updates. Consequently, their users would be deeply confused about the corresponding long sync delay though they may not notice the vast sync traffic.

Moreover, even the delta sync mechanism may also lead to *non-negligible* traffic overuse. For example, on handling the aforementioned data update pattern, the resulting Dropbox sync traffic is around 5.2 MB — much more than 1 MB (i.e., the total size of appended bytes) while much less than 260 MB or 186 MB (i.e., the sync traffic of GoogleDrive or 115 SyncDisk).

In this paper, we conduct a comparative study of various kinds of cloud sync mechanisms. For each mechanism, we mainly focus on the corresponding most representative cloud sync service. In order to better measure and understand the working principle of each mechanism, we artificially generate multiple special data update patterns. Thereby, we analyze their respective pros and cons and unravel the pathological processes for their traffic overuse problems.

Guided by the above knowledge, we provide insights/solutions for better choosing/designing a cloud sync service. For better choosing a cloud sync service, as a brief summary, in Fig. 1 we visualize the sync performances (mainly involving sync traffic and sync delay) of the investigated cloud sync mechanisms under different application situations. For better designing a cloud sync service, we propose the novel "*aTDS*" (*adaptive Timer-triggered Delta Sync*) mechanism, which adaptively tunes its timer threshold T_i to match the latest data update pattern (see Fig. 1) and thus has good performance under almost all the application situations. The efficacy of aTDS is confirmed by both our Linux-version prototype implementation and theoretical analysis.

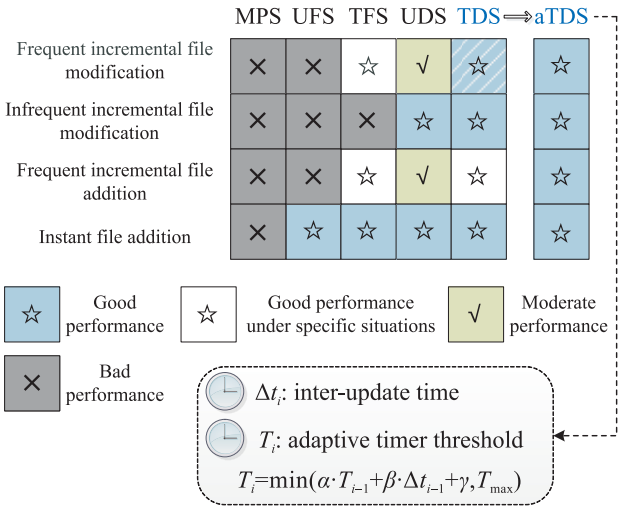

Fig. 1 Sync performances of the cloud sync mechanisms under different application situations. Here "frequent incremental file addition" means a novel file is added to the sync folder in an incremental manner.

2 Related Work

In the past few years, hundreds of cloud sync services have been deployed all over the world, making the topic of cloud sync service extremely hot and the relevant market increasingly competitive. In particular, the mainstream Internet companies (e.g., Google, Microsoft, Amazon, Apple, Baidu, and Dropbox) seem to be racing in the road of attracting and "sticking" users, mainly by optimizing the sync performances and thus enhancing the user experiences.

Four cloud sync services, i.e., Dropbox, Mozy, CrashPlan, and Carbonite, are studied in Ref. [26] from multiple perspectives including the backup (upload) and restore (download) performances, backup data types and restrictions, and so forth. The backup time is found to be tightly related to the compressibility of the file (because Mozy does not compress files before delivering them to the cloud), the amount of intra- and inter-account duplicate data, etc. On the other hand, restore is often faster than backup due to the client's download/upload bandwidth asymmetry. Among the four services, Dropbox behaves well in most aspects while Mozy behaves the worst.

Nevertheless, a large-scale passive measurement[27] of Dropbox reveals that its sync performance is mainly driven by the distance between the clients and Amazon S3, and short data updates coupled with a per-chunk acknowledgment mechanism lead to a bottleneck of transfer throughput. Moreover, an active

measurement[28] further reveals that the sync delay has become a potential performance bottleneck of Dropbox — the sync delay of Dropbox increases as the system scales, even often beyond the accepted level for practical collaboration.

A comparative tool called "CloudCmp"[29] is implemented to measure the performances of four "backend" cloud storage systems: Amazon AWS (including Amazon S3, SimpleDB, and SQS), Microsoft Azure, Google AppEngine, and Rackspace Cloud Servers. They are the supporting facilities of some popular cloud sync services, e.g., Dropbox employs Amazon S3 for its real data storage. Three major metrics, i.e., *operation response time*, *time to consistency*, and *cost per operation*, are used to evaluate the performance. It is discovered that each metric can vary significantly across different systems. In particular, Amazon S3 is found to be more fit for dealing with large data updates rather than small (incremental) data updates, which is consistent with our observation in Section 3.4. Besides, Bergen et al.[30] pointed out the client-side perceived performance primarily depends on the client's download bandwidth and the transfer bandwidth between the client and Amazon S3, rather than the upload bandwidth of the cloud. Consequently, designers of cloud sync services must pay special attention to the client-side sync performances and sync mechanisms.

There are still a number of issues that need to be addressed in designing a cloud sync (storage) service. For example, data segmentation is elaborated in Ref. [31], data cache/proxy is explored in Ref. [32], data consistency is investigated in Ref. [33], data de-duplication is studied in Refs. [34, 35], and some security and privacy issues are discussed in Refs. [26, 36]. These issues are out of the scope of this paper so we will not dwell on them.

3 Comparative Study of Cloud Sync Mechanisms

3.1 Preliminaries

In this section we conduct a comparative study of various kinds of cloud sync mechanisms. For each mechanism, we mainly focus on the corresponding most representative cloud sync service. All the experiments are performed by using the latest-version Windows client software (except UbuntuOne that uses a Linux client) as of August 2012 in Minneapolis, MN, US. All the client software runs on the same machine with a dual-core Intel CPU @2.26 GHz, a 2-GB RAM, and a 5400-RPM, 250-GB hard drive. Our Internet access bandwidth is around 4 Mbps ($\approx$ 500 KB/s). The communication packets involved in the cloud sync process are captured by using the "Wireshark" network protocol analyzer. For ease of reference, we list the major parameters (with regard to the cloud sync performance) as well as their basic explanations in Table 2.

Table 2 Parameter list.

Parameter	Description
n	Number of data updates involved in a cloud sync process.
t_i	Happening time of the i-th data update.
Δt_i	Inter-update time between the i-th data update and the (i+1)-th data update, that is to say, $\Delta t_i = t_{i+1} - t_i$.
$\overline{\Delta t}$	Average inter-update time.
T	Timer threshold used in a timer-triggered sync mechanism. When Δt_i is shorter than T, the i-th data update will be ignored; otherwise, the updated file or undelivered data updates is/are delivered to the cloud.
l_i	Length of the (compressed) i-th data update.
$\overline{l}$	Average (compressed) data update length.
$\lvert f_i\rvert$, $\lvert f_i'\rvert$, $\lvert\Delta f_i\rvert$	When the i-th data update happens, the local updated file is f_i and the corresponding cloud-stored file is f_i'. Δf_i is the "delta" between f_i and f_i'. Thus, $\lvert f_i\rvert$ is the size of f_i, $\lvert f_i'\rvert$ is the size of f_i', and $\lvert\Delta f_i\rvert$ is the size of Δf_i.
d_i	Sync delay of the i-th data update, that is, how much time the cloud sync client needs to fully deliver the i-th data update to the cloud since it happens (at t_i).
d_{total}	Total sync delay for all the data updates, that is, how much time the cloud sync client needs to fully deliver all the data updates to the cloud since the first (0-th) data update happens. Note that $d_{\text{total}} \neq \sum_{i=0}^{n-1} d_i$ in most cases.
m	Number of *data deliver events*. For most cloud sync services, not every data update can trigger a *data deliver event*, so $m \leqslant n$. Only in a data deliver event does the cloud sync client deliver an updated file or data updates to the cloud.
TR_j	Sync traffic incurred by the j-th *data deliver event*.
TR_{total}	Total sync traffic, that is, $\text{TR}_{\text{total}} = \sum_{j=0}^{m-1}(\text{TR}_j)$.
TR_{real}	Real data update traffic that denotes the real size of all the (compressed) data updates.
α	Ratio of the total sync traffic over the real data update traffic, that is, $\alpha = \dfrac{\text{TR}_{\text{total}}}{\text{TR}_{\text{real}}}$.

In order to better measure and understand the working principle of each mechanism, we artificially generate multiple special data update patterns. Some common patterns used for investigating each cloud sync mechanism are listed as follows:

• **1 MB-adding pattern** Adding a 1-MB RAR file to the sync folder, in order to examine the basic performance of the cloud sync mechanism on handling a simple instant data update.

• **1 byte-appending pattern** Appending 1 byte to an existing 1-MB RAR file in the sync folder, in order to recognize whether the cloud sync mechanism utilizes full-file sync or delta sync.

• ***X* KB/*X* s patterns** Appending X K random bytes to an existing empty file in the sync folder per X seconds until the total appended bytes reach 1 MB, where $X = 1, 2, 3, \cdots$, in order to figure out: (1) whether the cloud sync service uses an update-triggered sync mechanism or a timer-triggered sync mechanism, and (2) what is the timer threshold, T, if a timer is used. Thus, for each "XKB/Xs pattern", $n = \dfrac{1000}{X}$, $\Delta t_i = \overline{\Delta t} = X$ seconds, $l_i = \overline{l} = X$ KB, and $\mathrm{TR}_{\mathrm{real}}$ is kept as 1 MB to make the comparisons among these different patterns convenient. When we find $T \in (X, X+1)$, X is further tuned to $X.1, X.2, \cdots, X.9$ to figure out a more fine-grained timer threshold.

Thereby, we analyze their respective pros and cons under different application situations, and unravel the pathological processes for their traffic overuse problems.

3.2 UFS: Update-triggered Full-file Sync

UFS is utilized by 115 SyncDisk[1], UbuntuOne, Kanbox[4], Kuaipan[5], VDisk[6], and Wuala. In this subsection we focus on 115 SyncDisk since it is the most popular. First of all, we add a 1-MB RAR file to the 115 SyncDisk sync folder and we find that the resulting sync traffic is 1.07 MB and the resulting sync delay is 18 s[4]. Then we append 1 byte to this 1-MB RAR file, still resulting in 1.07 MB sync traffic and 18 s sync delay, which confirms that 115 SyncDisk employs a full-file sync mechanism.

Second, we append 1 K random bytes to an existing empty file in the 115 SyncDisk sync folder per second, until the total appended bytes reach 1 MB (thus, $\mathrm{TR}_{\mathrm{real}} = 1$ MB). The total sync traffic is $\mathrm{TR}_{\mathrm{total}} = 533$ MB (thus, $\alpha = \dfrac{\mathrm{TR}_{\mathrm{total}}}{\mathrm{TR}_{\mathrm{real}}} = 533$) and the total sync delay is d_{total}=2090 s. Obviously, although the total added/appended bytes are both 1 MB, the "1 KB/s pattern" brings much more sync traffic and much longer sync delay compared with the "1 MB-adding pattern", indicating that UFS leads to severe traffic overuse in the case of frequent incremental data updates. Furthermore, we examine the sync performances of 115 SyncDisk via multiple "X KB/X s patterns" where $X = 2, 3, 4, \cdots, 10$. The corresponding sync traffic and sync delay are plotted in Fig. 2, from which we can see that both $\mathrm{TR}_{\mathrm{total}}$ and d_{total} decrease as X increases but the traffic overuse is still significant.

Third, in order to unravel the pathological process for the above traffic overuse problem, we analyze the time series of the involved communication packets (mostly TCP packets). Figure 3 depicts the simplified working principle of UFS on handling frequent incremental data updates. Once a data update happens, the corresponding data deliver event is triggered (namely, $m \approx n$ in Table

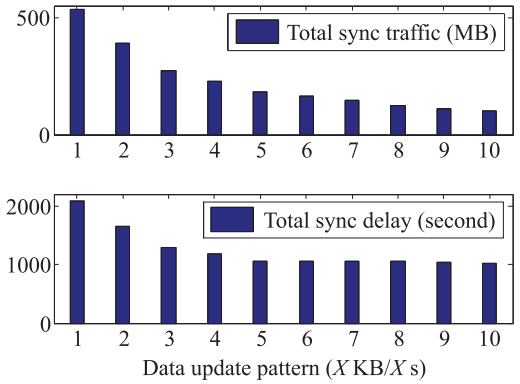

Fig. 2 Total sync traffic and sync delay of UFS (used by 115 SyncDisk) on handling "*X* KB/*X* s patterns".

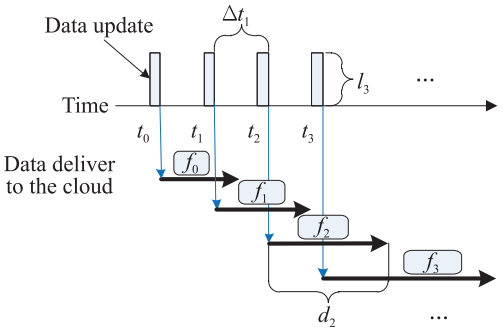

Fig. 3 Simplified working principle of UFS (used by 115 SyncDisk) on handling frequent incremental data updates.

[4]The 18 s' sync delay looks longer than our expectation because the 115 SyncDisk cloud is deployed in China while the experiment is performed in US. When we repeat the experiment in Beijing, China, the sync delay decreases to about 10 s.

2); thus, multiple data deliver events can compete for the user-side Internet access bandwidth at the same time, and we have $d_{\text{total}} \ll \sum_{i=0}^{n-1} d_i$. As for every data deliver event, the corresponding full updated file is delivered to the cloud and the cloud should replace the previous updated file with the latest one. Consequently, the abovementioned traffic overuse problem may also bring heavy network traffic and disk read/write burdens to the cloud.

3.3 TFS: Timer-triggered Full-file Sync

TFS is utilized by GoogleDrive, SkyDrive, Baidu CloudDisk[3], etc. In this subsection we focus on GoogleDrive since it is the most representative and is expected to be the most popular. Similar to 115 SyncDisk, adding a 1-MB RAR file to the GoogleDrive sync folder results in 1.11-MB sync traffic and 14-second sync delay. However, we observe that during the 14-second sync process, no communication packets are sent or received in the first 4 s, which reminds us that the data deliver event is *intentionally* delayed. Besides, appending 1 byte to this 1-MB RAR file also incurs 1.11-MB sync traffic and 14-second sync delay, indicating that GoogleDrive also employs a full-file sync mechanism.

The "1 KB/s pattern" brings very little traffic (mainly coming from periodical client-cloud beacon messages) to GoogleDrive before all the data updates are finished — very different from the case of 115 SyncDisk where $\text{TR}_{\text{total}} = 533$ MB. After the total 1-MB bytes are appended, the "real" sync process starts and the resulting sync traffic is nearly 1.2 MB. Moreover, we check the sync performances of GoogleDrive via multiple "X KB/X s patterns" where $X = 2, 3, 4, \cdots, 10$ and record their corresponding sync traffic and sync delay in Fig. 4. For $X = 2, 3$ the sync performances are similar to that of "1 KB/s pattern", but for $X = 4$ the sync traffic grows to 22 MB. When $X = 5$, the situation is totally different: the generated data updates continuously trigger data deliver events, and the resulting sync traffic is up to $\text{TR}_{\text{total}} = 260$ MB (thus, $\alpha = 260$) and the total sync delay is $d_{\text{total}} = 1870$ s. When $X > 5$, both TR_{total} and d_{total} decrease as X increases but the traffic overuse is always severe. As a result, we draw a conclusion that the timer threshold of GoogleDrive should be $T \in (3\,\text{s}, 5\,\text{s})$ and T is close to 4 s.

As illustrated in Fig. 5, the time series of GoogleDrive communication packets (corresponding to the "5 KB/5 s pattern") exhibit that $T \approx 4.2$ s with minor fluctuations in $(3.9\,\text{s}, 4.5\,\text{s})$. Apart from the timer, the basic working principle of GoogleDrive looks like that of 115 SyncDisk.

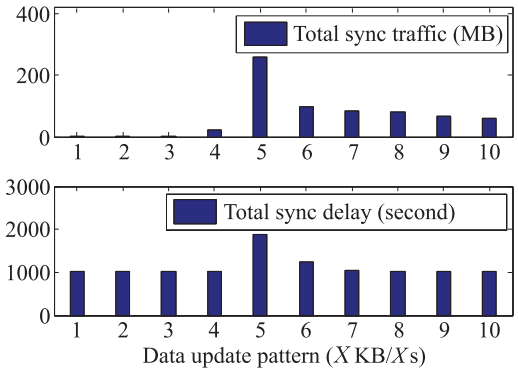

Fig. 4 Total sync traffic and sync delay of TFS (used by GoogleDrive) on handling "X KB/X s patterns".

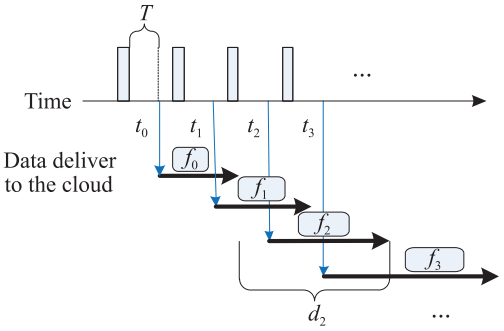

Fig. 5 Simplified working principle of TFS (used by GoogleDrive) on handling frequent incremental data updates.

3.4 UDS: Update-triggered Delta Sync

UDS is utilized by Dropbox, IDriveSync[7], 360 CloudDisk[8], Everbox[9], etc. In this subsection we focus on Dropbox since it is the most popular. When we add a 1-MB RAR file to the Dropbox sync folder, the resulting sync traffic is 1.22 MB and the resulting sync delay is 9.2 s. Then we append 1 byte to this 1-MB RAR file, resulting in 38.2-KB sync traffic and 4-second sync delay, which tells us that Dropbox employs a delta sync mechanism.

The "1 KB/s pattern" brings about 23-MB sync traffic to Dropbox, indicating that UDS leads to non-negligible traffic overuse in the case of frequent incremental data updates. Furthermore, we examine the sync performances of Dropbox by using multiple "X KB/X s patterns". The corresponding sync traffic and sync delay are plotted in Fig. 6, from which we have two observations: (1) the total sync traffic decreases as

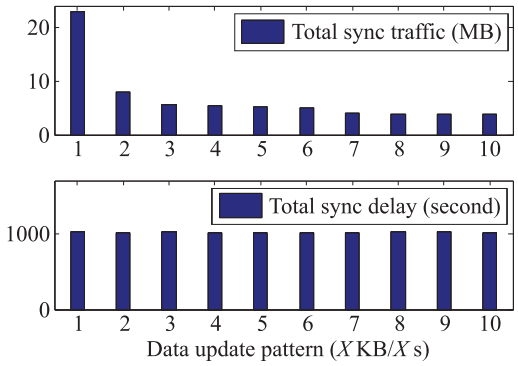

Fig. 6 Total sync traffic and sync delay of UDS (used by Dropbox) on handling "*X* KB/*X* s patterns".

X increases, and (2) the total sync delay stays stably between 1000 and 1020 s.

To explain the above two observations, we analyze the time series of the Dropbox communication packets and plot in Fig. 7 the simplified working principle of UDS on handling frequent incremental data updates. Once a data update happens, the Dropbox client first checks whether there is an on-going data deliver event — if yes, this data update will not trigger any data deliver event and may be merged to its following data update(s); otherwise, a data deliver event is triggered to upload the undelivered data update(s) to the "clouds". Here we use "clouds" rather than "cloud" because Dropbox hires Amazon's S3 cloud for its real data storage and meanwhile maintains a relatively small "Dropbox cloud" for metadata indexing, client-cloud beaconing, and so forth. As a result, it is quite possible that multiple Dropbox data updates are merged into one longer data update which then triggers one data deliver event, so we have $m < n$ when X is small and $m = n$ when X is large, but a larger X always implies a smaller m. As soon as all the data updates are finished, only one (i.e., the last) data deliver event is triggered to upload the last batch of undelivered data updates, so the total sync delay is always slightly longer than 1000 s.

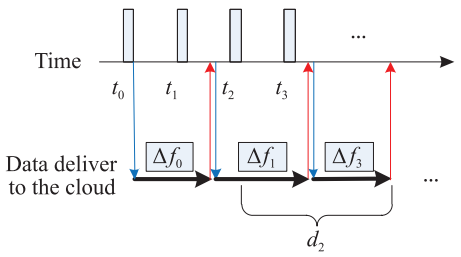

Fig. 7 Simplified working principle of UDS (used by Dropbox) on handling frequent incremental data updates.

3.5 TDS: Timer-triggered Delta Sync

Among the tens of popular cloud sync services we investigate, only SugarSync is found to utilize TDS. Adding a 1-MB RAR file to the SugarSync sync folder results in 1.07-MB sync traffic and 13-second sync delay. Besides, appending 1 byte to this 1-MB RAR file incurs about 60-KB sync traffic and 11-second sync delay, indicating that SugarSync also employs a delta sync mechanism.

Similar to GoogleDrive while different from Dropbox, the "1 KB/s pattern" brings very little traffic to SugarSync before all the data updates are finished. Moreover, we check the sync performances of SugarSync via multiple "X KB/X s patterns" where $X = 2, 3, 4, \cdots, 10$ and record their corresponding sync traffic and sync delay in Fig. 8. For $X = 2, 3, 4, 5$ the sync performances are the same with that of the "1 KB/s pattern", but for $X = 6$ the sync traffic sharply grows to 17.2 MB and for $X = 7$ the sync traffic reaches the maximum 33 MB, illustrating that the timer threshold (T) of SugarSync is close to 6 s. When $X > 7$, the sync traffic decreases as X increases but the traffic overuse is still non-negligible. On the other hand, the total sync delay is quite stable between 1000 and 1050 s.

As shown in Fig. 9, the basic working principle of TDS looks like that of UDS except for the timer. As a matter of fact, the sync process of SugarSync is slightly complicated than that depicted in Fig. 9. On handling a series of incremental data updates, SugarSync always delivers the first data update to the cloud without delay — in other words, the "timer" of SugarSync does not have effect on the first data update. Consequently, if the user generates just a single data update, he will see the sync process starts instantly and thus the user

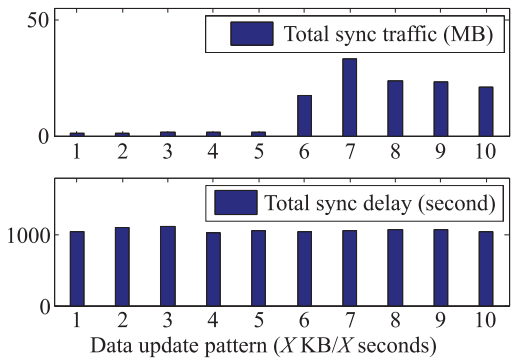

Fig. 8 Total sync traffic and sync delay of TDS (used by SugarSync) on handling "*X* KB/*X* s patterns".

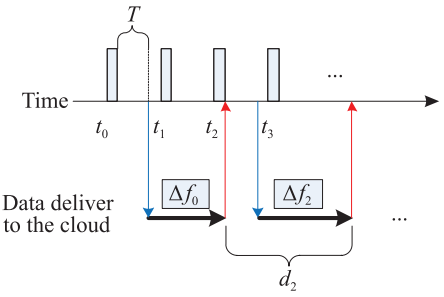

Fig. 9 Simplified working principle of TDS (used by SugarSync) on handling frequent incremental data updates.

experience is satisfactory. But for the second, third, and subsequent data updates, SugarSync will manage them according to its timer threshold.

3.6 MPS: Manual or Periodical Sync

MPS is utilized by Amazon CloudDrive, Box[2], IDrive, SpiderOak, CrashPlan, Evernote, Youdao Cloud Note[10], CloudMe[11], QQ Cloud Disk[12], DBank[13], etc. Considering Amazon's pioneering role in cloud computing and storage, we take Amazon CloudDrive as a typical example in this subsection. First, manually adding and then synchronizing a 1-MB RAR file to Amazon CloudDrive brings about 1.11-MB sync traffic and 10-second sync delay. Second, we append 1 byte to this 1-MB RAR file and manually synchronize it again to Amazon CloudDrive (to replace the old file), and the resulting sync traffic is still 1.11 MB and the sync delay is still 10 s. Therefore, Amazon CloudDrive employs a full-file sync mechanism. On the other hand, frequent incremental data updates usually have no effect on MPS because the local sync folder cannot be automatically and timely synchronized to the cloud by the client. Generally speaking, MPS is only suitable for applications that do not care about sync delay.

4 Insights/Solutions for Better Choosing/Designing a Cloud Sync Service

Guided by the knowledge obtained from the above comparative study, in this section we provide insights/solutions for better choosing/designing a cloud sync service.

4.1 Choosing an appropriate cloud sync service

For better choosing a cloud sync service, Fig. 1 has visualized the sync performances (mainly involving the sync traffic and sync delay) of UFS, TFS, UDS, TDS, and MPS under different application *situations*. Besides, we list more detailed application *scenarios* for the existing cloud sync mechanisms in Table 3. Here "*situations*" emphasize the data update patterns while "*scenarios*" emphasize the specific things, issues or services.

First of all, MPS is only suitable for those situations that do not care about sync delay, e.g., file backup. Although MPS might be the easiest to implement, its application scenario is too narrow and thus a common cloud sync service should try to avoid using MPS.

UFS is fit for instant file addition (e.g., picture/music/immutable file storing or sharing), rather than file modification (e.g., document editing and system log appending) or incremental file addition (e.g., file downloading). However, none of the popular cloud sync services (UbuntuOne and others in Refs. [1, 4, 5]) that utilize UFS have ever mentioned to their users about this. Thereby, their users would often edit a document in or download a file to the sync folder, and then be deeply confused about the resulting long sync delay (though they may not notice the vast sync traffic). In most cases, the sync performance of TFS is similar to that of UFS. Only in the case of a specific frequent incremental file addition does TFS work better than UFS, e.g., when GoogleDrive handles the "XKB/Xs patterns" with $X \leqslant 4$ (refer to Fig. 4).

UDS can well handle infrequent incremental file modifications like document editing and system log appending, because it only delivers the "delta" file to

Table 3 Appropriate application situations and scenarios for the existing cloud sync mechanisms.

Mechanism	App situations	App scenarios
MPS	Sync delay tolerant situations	⓪ File backup
UFS	Instant file addition	⓪ ① Picture/music/immutable file storing or sharing
TFS	Specific frequent incremental file addition	⓪① ② File downloading (partially)
UDS	Infrequent incremental file modification	⓪① ③ Document editing ④ System log append
TDS	Specific frequent incremental file modification	⓪①②③④ ⑤ Database updating (partially) ⑥ Sensor network data collecting (partially)

the cloud. But in the case of frequent incremental file modifications (e.g., database updating and sensor network data collecting), UDS would continuously upload numerous data updates to the cloud, resulting in abundant "*overhead traffic*" (including the traffic of DNS query, TCP/HTTP/HTTPS connection setup and maintenance, metadata retrieval, client-cloud beacon, etc.) that far exceeds the real size of data updates (namely, $\mathrm{TR}_{\mathrm{total}} \gg \mathrm{TR}_{\mathrm{real}}$). By moderately delaying the sync process of every data update, TDS can effectively overcome the abovementioned drawback of UDS *under specific situations* (compare Fig. 6 with Fig. 8). Unfortunately, among all the popular cloud sync services we have investigated in this paper, only SugarSync is found to utilize TDS, possibly because TDS is more complicated than others and is thus more difficult to implement.

4.2 aTDS: adaptive Timer-triggered Delta Sync mechanism

The above subsection reveals that none of the existing cloud sync mechanisms can well handle all the data update patterns. For better designing a cloud sync service, we propose the novel "*aTDS*" (*adaptive Timer-triggered Delta Sync*) mechanism, an enhanced version of TDS. Different from TDS that generally adopts a constant timer threshold T, aTDS adaptively tunes its timer threshold T_i to match the latest data update pattern and thus has good performance under almost all the application situations. Specifically, T_i is tuned in an *iterative* manner:

$$T_i = \min\left(\alpha \cdot T_{i-1} + \beta \cdot \Delta t_{i-1} + \gamma,\ T_{\max}\right) \quad (1)$$

where the weight constants $\alpha, \beta \in (0, 1)$ and $\alpha+\beta = 1$, γ is a small constant that guarantees T_i to be slightly longer than $\overline{\Delta t}$ in a small number of iteration steps[5], and $T_{\max}$ is also a manually configured constant denoting the upper bound of T_i. When the i-th data update happens (at t_i), we first get the latest inter-update time $\Delta t_{i-1} = t_i - t_{i-1}$, and then compute T_i according to Eq. (1). If the subsequent data update (i.e., the $(i+1)$-th data update) happens in T_i seconds (namely, $\Delta t_i < T_i$), the i-th data update will be ignored; otherwise, a novel data deliver event is triggered to upload the undelivered data updates to the cloud.

Typically, we set $\alpha = \beta = 0.5$ so that the "historical" information (T_{i-1}) and the "up-to-date" information (Δt_{i-1}) are equally weighted in computing T_i. For convenience, the small constant γ is also set as 0.5. Besides, $T_{\max}$ is configured as 10 s in order to restrict T_i within an acceptable range. Therefore, if SugarSync utilized aTDS rather than its original TDS with $T \approx 6$ s, on handling the "7 KB/7 s pattern", the resulting sync traffic would be close to 1 MB rather than the original 33 MB (refer to Fig. 8). More in detail, the t_i and T_i series should be like this:

$$t_0 = 0,\ t_1 = 7,\ t_2 = 14,\ t_3 = 21, \cdots, t_n = 7n;$$

$$T_0 = 0,\ T_1 = 4,\ T_2 = 6,\ T_3 = 7,\ T_4 = 7\frac{1}{2},\ T_5 = 7\frac{3}{4},\ T_6 = 7\frac{7}{8},\ T_7 = 7\frac{15}{16}, \cdots, T_n = 8 - \frac{1}{2^{n-3}}.$$

As a result, only the first four data updates can trigger separate data deliver events and all the other data updates will trigger only one data deliver event at last. Moreover, it is easy to discover that T_i can converge to slightly longer than the latest inter-update time in several iteration steps (which will be formally proved later).

To evaluate the practical efficacy of aTDS, we implement its Linux-version prototype by utilizing the open-source data synchronization tool rsync and the Linux kernel API inotify. rsync synchronizes files or folders from one location to the other in a "delta sync" manner, and inotify reports data updates (with regard to a file or a folder) to user applications in a "real time" manner. As depicted in Fig. 10, one client PC (locating at a common apartment in Minneapolis, US) and one cloud server (locating at the campus network of University of Minnesota in Minneapolis) are employed in our prototype implementation. First of all, a rsync daemon process regarding to the cloud-side sync folder is started and then keeps running in the cloud server, and aTDS invokes the system call inotify_add_watch(...) to monitor every data update regarding to the client-side sync folder. Then, aTDS can directly and instantly obtain the information of

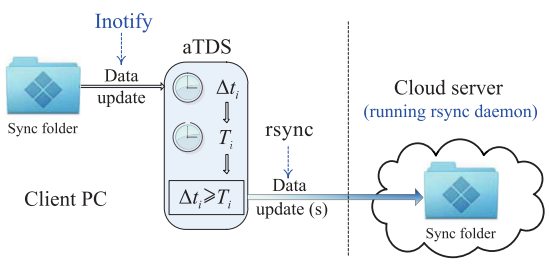

Fig. 10 Prototype implementation of aTDS.

[5]If we do not use γ (i.e., $\gamma = 0$), in many cases T_i can approach $\overline{\Delta t}$ but always stay below $\overline{\Delta t}$, and then aTDS has little effect in reducing the excessive sync traffic.

every data update — the information includes the path of the updated file, data update time, data update type (e.g., file modification, attribute modification, file creation and deletion), and so forth. Once a data update happens, its information is used by aTDS to compute Δt_i and T_i. If there is no subsequent data update in T_i seconds ($\Delta t_i \geqslant T_i$), the undelivered data updates are synchronized to the cloud server by using rsync.

When aTDS deals with the data update patterns described in Section 3.1, the "1 MB-adding pattern" results in 1.05-MB sync traffic and 6-second sync delay, and the "1 byte-appending pattern" incurs 12.7-KB sync traffic and nearly 2-second sync delay. Besides, as plotted in Fig. 11, each "XKB/Xs pattern" ($X = 1, 2, 3, \cdots, 10$) brings less than 1.1-MB sync traffic and the total sync delay is very close to 1000 s.

In practice, the data update pattern is usually more complicated than an "XKB/Xs pattern" and often exhibits a "hybrid" inter-update time distribution (e.g., as depicted in Fig. 12). Then we can formally prove that aTDS can still effectively reduce the number of data deliver events and thus reduce the sync traffic.

Theorem 1 Suppose a hybrid data update pattern P consists of m basic data update patterns and each basic pattern P_i ($i = 0, 1, 2, \cdots, m-1$) is composed of n_i data updates with the same inter-update time τ_i (usually, $\tau_i < T_{\max}$). When aTDS is applied, at most $\sum_{i=0}^{m-1} \log(\tau_i + 1)$ data deliver events will be triggered.

Proof Let $T_{i,k}$ denote the k-th timer threshold in the i-th basic data update pattern. First, we consider P_0 that comprises n_0 data updates with the inter-update time τ_0. According to Eq. (1) and because $\tau_0 < T_{\max}$ (i.e., $\tau_0 + 1 \leqslant T_{\max}$), we have the following $T_{0,k}$ series:

$$T_{0,0} = 0,\ T_{0,1} = \frac{1}{2}(\tau_0 + 1),\ T_{0,2} = \frac{3}{4}(\tau_0 + 1),$$
$$T_{0,3} = \frac{7}{8}(\tau_0 + 1),\ T_{0,4} = \frac{15}{16}(\tau_0 + 1),\ \cdots,$$
$$T_{0,k} = (1 - \frac{1}{2^k})(\tau_0 + 1),\ \cdots,$$
$$T_{0,n_0-1} = (1 - \frac{1}{2^{n_0-1}})(\tau_0 + 1).$$

Since a data deliver event is triggered only when $T_{0,k} \leqslant \tau_0$, that is $(1-\frac{1}{2^k})(\tau_0+1) \leqslant \tau_0$, we get $2^k \leqslant \tau_0+1$. As a result, $k \leqslant \log(\tau_0+1)$, that is to say, P_0 can trigger at most $k_0 = \log(\tau_0 + 1)$ data deliver events — note that k_0 is independent of the number of data updates (n_0) contained in P_0.

Next, we consider P_1 that comprises n_1 data updates with the inter-update time τ_1 ($\tau_1 \neq \tau_0$). Similarly, we have the following $T_{1,k}$ series for P_1:

$$T_{1,0} = \frac{T_{0,n_0-1}}{2} + \frac{1}{2}(\tau_1 + 1),\ T_{1,1} = \frac{T_{0,n_0-1}}{4} + \frac{3}{4}(\tau_1 + 1), \cdots,\ T_{1,k} = \frac{T_{0,n_0-1}}{2^{k+1}} + (1 - \frac{1}{2^{k+1}})(\tau_1 + 1),\ \cdots$$

Since $T_{0,n_0-1} \approx \tau_0 + 1$ and let $T_{1,k} \leqslant \tau_1$, we get $2^{k+1} \leqslant \tau_1 - \tau_0$. If $\tau_1 < \tau_0$, k has no solution, which means no data deliver event is triggered; otherwise, $k \leqslant \log(\tau_1 - \tau_0) - 1 < \log(\tau_1 + 1)$. In general, P_1 can trigger at most $k_1 = \log(\tau_1 + 1)$ data deliver events.

In the same way, P_i can trigger at most $k_i = \log(\tau_i + 1)$ data deliver events; therefore, when aTDS is applied, at most $\sum_{i=0}^{m-1} k_i = \sum_{i=0}^{m-1} \log(\tau_i + 1)$ data deliver events will be triggered. ■

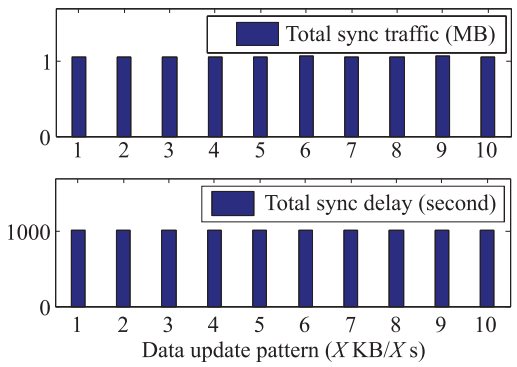

Fig. 11 Sync traffic and delay of aTDS on handling "X KB/X s patterns".

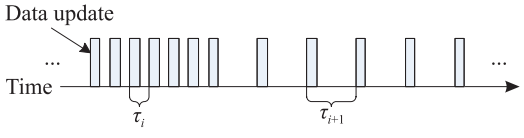

Fig. 12 A "hybrid" data update pattern.

5 Conclusions and Future Work

Cloud sync service represents a new paradigm of Internet-based data storing/sharing and people have witnessed its quick and great success in industry. Delving into tens of popular cloud sync services, we find their kernel component, i.e., the cloud sync mechanism, plays a critical role in cloud sync services because it greatly affects the cloud operation cost and user experience. Then we further figure out that the existing cloud sync mechanisms can be generally classified into five categories: UFS, TFS, UDS, TDS, and MPS.

To obtain an in-depth understanding of these mechanisms, in this paper we artificially generate multiple special data update patterns to measure their

working performances (including sync traffic, sync delay, etc.), illustrate their working principles, and analyze their respective pros and cons. In particular, we discover that even some very popular cloud sync services utilize a quite coarse-grained cloud sync mechanism that may lead to severe traffic overuse. We unravel the pathological processes for their traffic overuse problems and propose the novel "aTDS" cloud sync mechanism (an enhanced version of TDS) that has good performance under almost all the application situations. The efficacy of aTDS is confirmed by both our prototype implementation and theoretical analysis.

Still some future work remains. First, this paper focuses on the "data-upload" cloud sync mechanism (i.e., how to deliver the user-side data updates to the cloud) rather than the "data-download" cloud sync mechanism, because upload related operations (like file addition and modification) are usually considered to happen much more frequently than download related operations (like file sharing). Investigation of the "data-download" cloud sync mechanism may reveal more interesting issues and valuable problems.

Second, for each cloud sync mechanism, we mainly focus on the corresponding most representative cloud sync service, and thus do not elaborate other services as well as their mutual similarities and distinctions. For example, both 115 SyncDisk and UbuntuOne utilize the UFS mechanism but their concrete sync processes and sync performances still have some differences. On handling the "1 KB/s pattern", 115 SyncDisk results in 533-MB sync traffic and 2090-second sync delay while UbuntuOne incurs 166-MB sync traffic and 1018-second sync delay, because every data update triggers a data deliver event as for 115 SyncDisk while several data updates are merged to trigger a data deliver event as for UbuntuOne.

Third, as to each cloud sync service, only the sync performance and working principle of its PC client is studied. As a matter of fact, many cloud sync services also have their mobile clients that run on iPhone, iPad, Android or Blackberry devices. For example, the Dropbox PC client uses the UDS mechanism but its Android client uses the MPS mechanism so as to minimize the sync traffic. Furthermore, most cloud sync services provide a Web-version user interface which can be operated via a common Web browser. We believe there remains considerable optimization space in designing the mobile-version and Web-version cloud sync services.

Finally, our aTDS prototype is implemented on top of the Ubuntu Linux operating system but most popular cloud sync clients run in the Windows environment. Although many cloud sync services have developed both Linux and Windows clients, the latter are usually much more than the former. Therefore, we plan to implement the Windows-version (and even the MacOS-version) aTDS in the future.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (No. 61073015), the National Key Basic Research and Development (973) Program of China (No. 2011CB302305), and National Key Projects of Science and Technology of China (No. 2010ZX03004-001-03).

References

[1] 115 SyncDisk, http://pc.115.com/box, 2012.
[2] Box.com, http://box.com, 2012.
[3] Baidu CloudDisk, http://pan.baidu.com, 2012.
[4] Kanbox, http://www.kanbox.com, 2012.
[5] Kingsoft Kuaipan, http://www.kuaipan.cn, 2012.
[6] VDisk, http://vdisk.me, 2012.
[7] IDriveSync, http://www.idrivesync.com, 2012.
[8] 360 CloudDisk, http://yunpan.360.cn, 2012.
[9] Everbox, http://www.everbox.com, 2012.
[10] Youdao Cloud Note, http://note.youdao.com, 2012.
[11] CloudMe, http://www.cloudme.com, 2012.
[12] QQ Cloud Disk, http://disk.qq.com, 2012.
[13] Huawei DBank, http://www.dbank.com, 2012.
[14] Dropbox is now the data fabric tying together devices for 100M registered users who save 1B files a day, http://techcrunch.com/2012/11/13/dropbox-100-million, 2012.
[15] The number of 115-NetDisk users has exceeded 30M, http://www.donews.com/net/201203/1139233.shtm, 2012.
[16] GoogleDocs, http://docs.google.com, 2012.
[17] E. Zohar, I. Cidon, and O. Mokryn, The power of prediction: Cloud bandwidth and cost reduction, in *Proc. 2011 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, Toronto, Canada, 2011.
[18] A. Tridgell and P. Mackerras, The rsync algorithm, Computer Science Technical Report Series TR-CS-96-05, Australian National University, Australia, 1996.
[19] Binary delta compression, http://en.wikipedia.org/wiki/Binary_delta_compression, 2012.
[20] Binary diff wiki, http://en.wikipedia.org/wiki/Diff, 2012.
[21] DropboxTeams, http://www.dropbox.com/teams, 2012.
[22] Dropbox as database: Tutorial, http://blog.opalang.org/2012/11/dropbox-as-database-tutorial.html, 2012.
[23] Tutorial: Use Dropbox to share a database or any file, http://www.myquerybuilder.com/blog/2011/09/01/tutorial-use-dropbox-to-share-a-database-or-any-file, 2012.

[24] Y. He and Y. Liu, VOVO: VCR-oriented video-on-demand in large-scale peer-to-peer networks, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 20, no. 4, pp. 528-539, 2009.

[25] X. Liao, H. Jin, Y. Liu, and L. Ni, Scalable live streaming service based on inter-overlay optimization, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 18, no. 12, pp. 1663-1674, 2007.

[26] W. Hu, T. Yang, and J. N. Matthews, The good, the bad and the ugly of consumer cloud storage, *ACM SIGOPS Operating System Review*, vol. 44, no.3, pp. 110-115, 2010.

[27] I. Drago, M. Mellia, M. Munaf, A. Sperotto, R. Sadre, and A. Pras, Inside dropbox: Understanding personal cloud storage services, in *Proc. 12th ACM SIGCOMM Internet Measurement Conference (IMC)*, Boston, MA, USA, 2012.

[28] H. Wang, R. Shea, F. Wang, and J. Liu, On the impact of virtualization on dropbox-like cloud file storage/synchronization services, in *Proc. 20th IEEE/ACM Workshop on Quality of Service (IWQoS)*, Coimbra, Portugal, 2012.

[29] A. Li, X. Yang, S. Kandula, and M. Zhang, CloudCmp: Comparing public cloud providers, in *Proc. 10th ACM SIGCOMM Internet Measurement Conference (IMC)*, Melbourne, Australia, 2010.

[30] A. Bergen, Y. Coady, and R. McGeer, Client bandwidth: The forgotten metric of online storage providers, in *Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim)*, Victoria, B.C., Canada, 2011.

[31] M. Vrable, S. Savage, and G. Voelker, Cumulus: Filesystem backup to the cloud, in *Proc. 7th USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, USA, 2009.

[32] M. Vrable, S. Savage, and G. Voelker, BlueSky: A cloud-backed file system for the enterprise, in *Proc. 10th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, USA, 2012.

[33] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. Mckelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Fahim ul Haq, M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, Windows azure storage: A highly available cloud storage service with strong consistency, in *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[34] D. Harnik, B. Pinkas, and A. Shulman-Peleg, Side channels in cloud services: Deduplication in cloud storage, *IEEE Security & Privacy*, vol. 8, no. 6, pp. 40-47, 2010.

[35] S. Halevi, D. Harnik, B. Pinkas, and A. Shulman-Peleg, Proofs of ownership in remote storage systems, in *Proc. 18th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, 2011.

[36] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl, Dark clouds on the horizon: Using cloud storage as attack vector and online slack space, in *Proc. 20th USENIX Security Symposium*, San Francisco, CA, USA, 2011.

Zhenhua Li is a PhD candidate in computer science and technology at Peking University, Beijing, China. He was also a joint PhD student in computer science and engineering at the University of Minnesota — Twin Cities, USA. His current research areas mainly consist of cloud computing/storage, Internet content distribution, and peer-to-peer technologies. He has published one book and over 25 technical papers in the above areas. He is a member of the ACM, the ACM SIGMM, the IEEE-CS, (Computer Society) and the CCF (China Computer Federation).

Zhi-Li Zhang received the BS degree from Nanjing University, Jiangsu, China, in 1986, and the MS and PhD degrees from the University of Massachusetts, Amherst, in 1992 and 1997, respectively, all in computer science. In 1997, he joined the Computer Science and Engineering faculty with the University of Minnesota, Minneapolis, where he is currently a professor. From 1987 to 1990, he conducted research with the Computer Science Department, Aarhus University, Aarhus, Denmark, under a fellowship from the Chinese National Committee for Education. He has held visiting positions with Sprint Advanced Technology Labs, Burlingame, CA; IBM T. J. Watson Research Center, Yorktown Heights, NY; Fujitsu Labs of America, Sunnyvale, CA; Narus Inc., Microsoft Research; INRIA, Sophia-Antipolis, France; Universidad de Carlos III de Madrid and IMDEA Networks. He is a co-recipient of three Best Paper Awards from ACM SIGMETRICS, IEEE ICNP, and IEEE INFOCOM.

Yafei Dai is a professor at the Department of Computer Science and Technology, Peking University, Beijing, China. She received her PhD degree in computer science and technology at the Harbin Institute of Technology, China. Her research areas mainly include networked and distributed systems, P2P computing, network storage, and online social networks. She is a member of the IEEE, the IEEE Computer Society, and the CCF (China Computer Federation).