This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCOMM.2017.2731848, IEEE Communications Letters

RSD: Rate-based Sync Deferment for Personal Cloud Storage Services

Raúl Sáiz-Laudó, Marc Sánchez-Artigas, *Member, IEEE*, Pedro García-López, *Member, IEEE*

***Abstract*—Cloud storage services like Dropbox, Google Drive, and OneDrive, to cite a few, are becoming an increasingly "vital" tool in our everyday life. Unluckily, these services can incur large network overhead in different usage scenarios. To reduce it, these systems utilize several techniques like source-based deduplication, chunking, delta compression, etc. One of these techniques is sync deferment, which relies on the packing of updates to intentionally defer the synchronization process for some time, and increase the volume of useful data per overhead byte. The scientific literature has shown this technique to be very helpful, though there are still some limitations on current solutions. To resolve them, we present here a new adaptive sync deferment method, that is comparable to the current state of the art in terms of network overhead, but is also able to minimize the file synchronization time up to 12X.**

***Index Terms*—Personal Cloud storage, file sync deferment.**

I. Introduction

As a tool for personal storage and file synchronization, cloud storage services like Dropbox, OneDrive or Google Drive have become part of our everyday life. In these systems, the process of synchronizing a file requires of several metadata and data transfers between the cloud servers and the end user devices. To reduce the network overhead, these services use a suite of tools like delta compression, chunking, etc. [1], [2].

One of these optimizations, and the subject of this letter, is *sync deferment*. Sync deferment consists of batching updates to intentionally defer the sync process for some time. In this way, the client can artificially increase the amount of useful data per sync operation, and hence, diminish the network overhead [1]. This has proven to be very effective to cope with frequent file modifications [3]. The authors of [1], however, discovered that simple sync deferment based on static thresholds could be not helpful. To wit, they found that by using a *fixed* sync deferment time (not tunable by the user), the network overhead could be of several orders of magnitude larger than the amount of useful data in some situations. To address this issue, they proposed an *adaptive sync deferment* (ASD) technique that adjusts the sync deferment time based on the *inter-update time*.

As a part of our ongoing process of implementing an open-source cloud storage service based on [2], we carefully studied the ASD algorithm, and found that in some cases, it may render long synchronization delays. Such a finding spurred us to come up with a novel deferment algorithm that we contribute here. While it is comparable to ASD in terms of network overhead, its distinguishing feature is that it is also capable of *minimizing the synchronization delay*, up to 12X compared with ASD. This translates into a better user experience, and less conflicts. The reason is that we study another dimension beyond inter-update time: *the data size of updates*. Combining both metrics, we can operate with the data rate instead, and adjust the deferral time to the minimum necessary to presumably collect enough useful data, and thus deliver a low network overhead.

R . Sáiz-Laudó , M. Sánchez-Artigas, P. García-López are with the Dept. of Computer Engineering and Maths, Universitat Rovira i Virgili, Spain. (e-mail: {raul.saiz, marc.sanchez}@urv.cat).

This work has been partially funded by the Spanish MEyC (grant TIN2013-47245-C2-2-R) and by EU (project IOStack, grant H2020-ICT-2014-7-1).

II. Background

A. Network overhead

Throughout the paper, we will utilize the TUE metric [1] to quantify the network overhead. TUE stands for ***T**raffic **U**sage **E**fficiency*, which is defined as:

$$TUE = \frac{\text{Total sync traffic}}{\text{Data update size}},$$

where the *data update size* refers to the size of the altered bits in the update due to file creation, modification or removal. It intuitively signals the network overhead from the user side.

B. Adaptive Sync Deferment

We describe the *adaptive sync deferment* (ASD) algorithm presented in [1]. To the best of our knowledge, ASD is the only algorithm in the literature that *adaptively* adjusts the deferment time. We briefly revisit it here. It works as follows:

Upon the ith update at time t_i, the idea behind ASD is to adaptively tune the sync deferment time window T_i, such that if the next update falls within the range t_i to $t_i + T_i$, then it is deferred and marked as pending in the client. Otherwise, all the pending updates are pushed to the cloud backend. Specifically, T_i is tuned in an iterative manner as an EWMA (***E**xponentially **W**eighted **M**oving **A**verage*) controller:

$$T_i = \min\left((1-\omega)T_{i-1} + \omega\Delta t_i + \epsilon, T_{max}\right), \qquad (1)$$

where Δt_i is the inter-update time between the (i-1)th and the ith data updates, and $\epsilon \in (0,1.0)$ is a small constant that ensures T_i to be slightly longer than Δt_i in a small number of iteration rounds. T_{max} is a constant representing an upper bound on T_i, to prevent a large T_i from harming user experience due to long sync delays.

By a simple inspection of (1), it is easy to see that ASD does not account for the size of updates. It focus only on one single dimension: *inter-update time*. Although this metric is sufficient to reduce the TUE, it does not always minimize the sync delay.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCOMM.2017.2731848, IEEE Communications Letters

III. Rate-based sync deferment

Although it has been seen in [1] that ASD outperforms sync deferment algorithms based on static values (e.g., the number of uncommitted bytes), it only considers an EWMA estimator of the inter-update time as the deferment criteria. And hence, it neglects an important dimension: *the size of updates*. The TUE metric, however, measures the traffic overhead. Consequently, what should be key for a sync deferment scheme should be to accumulate a sufficiently large number of unsynced bytes such that TUE was close to 1, yet delivering a short synchornization delay. With time alone, it is very difficult to achieve this, as we demonstrate in the evaluation. It is also necessary to consider the count of deferred bytes. Put another way, when the number of deferred bytes guarantees a small TUE, it does not yet make sense to wait for a new update, but to trigger a sync operation with the cloud backend as soon as possible.

A. Trade-off between TUE and synchronization time

It is clear that a sync deferment algorithm able to hold good levels on both parameters is still missing. This task is not easy. There is a trade-off between both parameters: when one tries to minimize one dimension, the other can grow uncontrollably.

To better understand this, pretend now that the dimension to optimize is the synchronization delay. Clearly, this will benefit user experience, since any modification to a file will be quickly propagated to the unsynced devices. However, it will impose a huge overhead on the system, because the count of unsynced bytes will be typically small. On the other hand, suppose now that objective is to decrease the network overhead. This would require deferring updates in order to transmit more useful bytes per overhead byte. Depending on to what extent, however, the synchronization delay could become intolerably long, and for instance, preclude services such as collaborative file editing.

To give a sense of this trade-off, we investigated the impact of sync deferment delay on TUE in Ubuntu One (UB1), a real cloud storage service. Concretely, we randomly picked $10,000$ client sessions from the publicly available trace [4]. Then, for each client session, we recorded the resulting TUE obtained by varying the sync deferment threshold from 30 to 120 seconds. The resulting TUE values were averaged to produce Fig. 1.

Since the UB1 service shut down on July 2014, we could not use its desktop client to empirically measure the TUE. Instead, to approximate the resulting TUE as a function of the amount of deferred data, we performed a small measurement analysis of Dropbox similar to that in [1]. The idea was pretty simple. To approximate its real TUE, we measured the resulting TUE when adding files of different sizes to the sync folder. To avoid any bias, we used binary, *non-compressible* files, ranging from 1 B to 100 MB, so that the TUE can be simply approximated as the ratio between the monitored network traffic after every file addition and the file size. Actually, we got similar results to those listed in [1], [3], with a TUE of ≈ 37 for 1 KB files, and of ≈ 1.1 for 100 MB files. As a result of this measurement, we could return a TUE value for every potential size of deferred data by means of curve interpolation, and thus produce Fig. 1. We also used the interpolated TUE curve in our evaluation.[1].

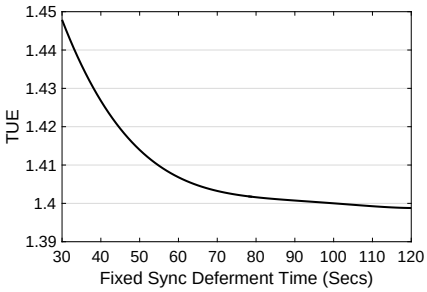

Fig. 1. TUE as a function of sync deferment time in UB1.

As can be seen in this figure, the longer the client waits to send the pending updates to the cloud, the shorter the TUE is. The important observation to be made is that deferring updates too much is not of much utility. Beyond a certain point, TUE does not reduce significantly. This suggests that *as soon as the size of the deferred updates yields a small TUE, it is better off to push them to the cloud.* Almost surely, waiting for the next update will bring no much benefit. ASD cannot exploit such a trade-off well. For this reason, we devised the RSD algorithm.

B. The RSD algorithm

Given a target network overhead $\overline{TUE}$, our major objective is to minimize the synchronization time. To this end, instead of adapting the sync deferment time according to the inter-update time as in ASD, we do so by turning attention onto the *update rate*, defined as $R = b/\Delta t$, where b is the total number of bytes accumulated from local updates over certain time interval Δt. By estimating the rate R, we can dynamically adjust the sync deferment delay to the data generation rate of a user, so that it can be shortened if it is expected that the targeted $\overline{TUE}$ will be fulfilled soon according to the predicted rate. This is the reason why our algorithm is called *rate-based sync deferment* (RSD).

In practice, our scheme expresses the overhead objective as the number of unsynced bytes necessary to accumulate in order to yield an overhead equal or lower than $\overline{TUE}$. We denote this quantity as $B^{\overline{TUE}}$. Note that $B^{\overline{TUE}} = (\text{Total sync traffic})/\overline{TUE}$.

To estimate the rate, RSD utilizes an EWMA predictor. This means that upon the ith update, RSD estimates the current rate R_i as an average between the value of the last estimation R_{i-1}, and the current observation $b_i/\Delta t_i$ such that:

$$R_i = (1-\omega)R_{i-i} + \omega\frac{b_i}{\Delta t_i}, \tag{2}$$

where Δt_i is the inter-update time between the (i-1)th and the ith data updates, b_i is the size of the ith update in bytes, and ω is the weighting factor that shapes its memory. With the value of R_i at hand, then RSD calculates the sync deferment time T_i as $T_i = \frac{B^{\overline{TUE}} - B^{\text{ACC}}}{R_i}$, i.e., as the number of bytes still needed to amass from future updates ($B^{\overline{TUE}} - B^{\text{ACC}}$) divided by R_i, where B^{ACC} is a byte counter that tracks the size of all updates since

[1]All the experiments in this letter were performed on a commodity machine: Intel Core i5-4440 3.10GHz CPU, 8 GB RAM, connected to a 1 GbE LAN.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCOMM.2017.2731848, IEEE Communications Letters

the last sync operation with the cloud backend. After syncing the pending updates, B^{ACC} is always reset to 0.

As in ASD, a sync operation is started with the cloud servers when the next update falls outside the range t_i to $t_i + T_i$. Also, to ensure that updates do not remain unsynced for a long time, RSD uses a timer $\mathscr{T}$. When $\mathscr{T}$ expires, a new sync operation is triggered, irrespective of whether the target $\overline{TUE}$ is met or not. Note that this is different from limiting the maximum allowed value for each individual T_i as in ASD. Indeed, ASD does not ensure that the deferred updates are eventually applied. This is easy to see by simply inspecting (1). If inter-update times were always $< T_{max}$ (e.g., as a result of a frequent file modification), ASD could defer the updates forever if EWMA converged to a stationary value. In RSD, we addressed this issue from the start by using an independent timer, which is re-programmed upon the arrival of the first update after the last sync operation. The complete pseud code is listed in Algorithm 1.

Algorithm 1 RSD algorithm

upon ith update happens **do**
  if $B^{\text{ACC}} = 0$ **then**
    set timer $\mathscr{T}$ to T_{max} seconds
  $\Delta t_i := t_i - t_{i-1}$ ▷ compute the inter-update time
  $R_i := (1-\omega)R_{i-i} + \omega\frac{b_i}{\Delta t_i}$ ▷ update the EWMA estimator of the rate
  $B^{\text{ACC}} := B^{\text{ACC}} + b_i$
  if $T_{i-1} < \Delta t_i$ **then**
    push the deferred B^{ACC} bytes to the cloud; $B^{\text{ACC}} := 0$
  else
    $T_i := \frac{B^{\overline{TUE}} - B^{\text{ACC}}}{R_i}$
upon timer $\mathscr{T}$ expires **do**
  push the deferred B^{ACC} bytes to the cloud; $B^{\text{ACC}} := 0$

C. Analytical Comparison: RSD vs. ASD

Here we compare analytically the performance of RSD with ASD to better understand the benefits of sync deferment based on the data generation rate. For this purpose, we will adopt the "X KB/X sec" pattern that was originally posited in [1], [5] to validate ASD. As in [5], we will set the weight factor $\omega = 1/2$ to simplify our discussion. From the analysis in [5], it follows easily the following corollary:

Corollary 1: Under the "X KB/X sec" pattern, ASD *triggers at most* $k = \lg X + 1$ *data transfers to the cloud.*

Indeed, it is easy to see that these data transfers occur at the first k updates, i.e., while the value of T_i converges to X. From that point onwards, (1) guarantees that $T_i > X$, for $i > k$[2], and hence, subsequent file updates may stay in pending state at the desktop client for a long time, even forever, yielding very long synchronization times.

With RSD, however, the value of T_i decreases progressively towards 0, to ensure that when the size of the deferred updates reaches $B^{\overline{TUE}}$ bytes, a sync operation with the cloud backend is always started. This minimizes the synchronization time yet delivering a small TUE:

[2]Under the "X KB/X sec" pattern, the inter-update time is $\Delta t_i = X$ sec $\forall i$.

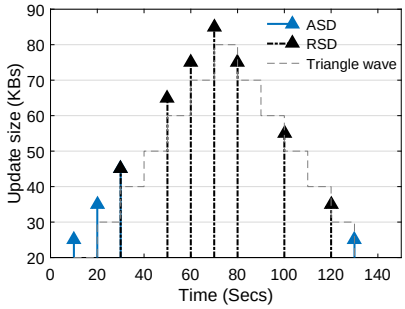

Fig. 2. ASD vs. RSD under a regular triangular pattern.

Theorem 1: Under the "X KB/X sec" pattern, RSD *delivers the minimum synchronization delay to satisfy* $\overline{TUE}$.

Proof: Let $B^{\overline{TUE}}$ denote the number of bytes necessary to accumulate to meet the objective $\overline{TUE}$. To verify that RSD minimizes the synchronization time, we must prove that there is no data transfer to the cloud until the target $\overline{TUE}$ is fulfilled. First, we note that the observed rate $r_i = \frac{X}{X} = 1$ for all updates. Similar to [5], note that we assume $B^{\overline{TUE}} < T_{max}$. Otherwise, a sync operation would be triggered without reaching the target $\overline{TUE}$. Then, according to (2), we will have the series $R_k = \left(1 - \frac{1}{2^k}\right)$, which gives the corresponding time window series:

$$T_k = \frac{B^{\overline{TUE}} - kX}{R_k} = \left(B^{\overline{TUE}} - kX\right)\left(\frac{2^k}{2^k - 1}\right). \qquad (3)$$

Recall that a sync operation is triggered only if $T_k < X$ (where X is the inter-update time). From (3), it is easy to see that this inequality does not hold for $k \leq \frac{B^{\overline{TUE}}}{X} - 1$. Hence, a new sync operation with the cloud will not be started until at least $B^{\overline{TUE}}$ bytes are accumulated from the deferred updates, thus meeting the target $\overline{TUE}$ while yielding the minimum sync delay. ■

To sum up, under a regular pattern, RSD is able to minimize the synchronization time, triggering a new sync operation with the cloud only when the overhead is optimal. In contrast, ASD triggers a logarithmic number of sync operations until reaching a stable state, and then, it defers updates forever until there is a significant change in the inter-update time. This behavior is easy to see in the triangular pattern shown in Fig. 2, where the bitsize of updates increases gradually up to 80KBs to decrease with the same speed. While RSD triggers a new sync operation as soon as $B^{\overline{TUE}}$ bytes has been buffered, spreading them over time, ASD only triggers a final sync operation after adjustment of the deferral time to the regular inter-update time, remaining out-of-sync for 110 seconds.

IV. Experimental Comparison

In practice, however, data update patterns are not so regular as the "X KB/X sec" pattern. For this reason, we compared the performance of both algorithms using real sessions from UB1 users [4]. From this trace[3], and after some pre-processing, we extracted 7 random workloads[4] corresponding to 7 different

[3]The UB1 log trace contains the timestamp and the size of every update, among other information.

[4]The workloads are available upon request.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LCOMM.2017.2731848, IEEE Communications Letters

TABLE I
DETAILS OF WORKLOADS

		Inter-update time (secs)		Update size (MB)	
Workload	No. Updates	90%-tile	Skewness	Median	CV
1	1, 653	16	3.24	2.74	1.41
2	5, 133	1	30.12	0.03	6.19
3	3, 463	7	18.2	0.40	4.53
4	736	61	1.02	0.81	0.28
5	505	49	3.13	5.54	0.67
6	441	234.6	1.96	5.01	0.67
7	730	125.1	–0.11	0.01	0.51

users. For clarity, we numbered them from 1 to 7. Workloads $1-5$ corresponded to active users with inter-updates times of a few seconds. Their purpose was to evaluate both algorithms in the face of frequent of file modifications, which is the major driving force for sync deferment techniques. Workloads 6 and 7 corresponded to "warm" users, and concretely, to users who often made changes to their sync folder with a frequency that exceeded 120 sec. The goal of the last two workloads was to evaluate the performance of RSD when the timer $\mathscr{T}$ expires. More details about workloads can be found in Table I.

Metrics. Besides TUE, we compared both algorithms in terms of synchronization delay, which is the aspect that distinguishes RSD from ASD. To this aim, we utilized the slowdown ratio SR defined as $SR = \frac{Time_{\text{ASD}}}{Time_{\text{RSD}}}$, where $Time_{\text{ASD}}$ and $Time_{\text{RSD}}$ are the average lengths of deferment periods delivered by ASD and RSD, respectively. Note that $SR = 1$ if both algorithms perform identically. A value of $SR > 1$ quantifies how many times the synchronization delay is larger in ASD compared with RSD.

Setup. The experimental setup was as follows. In both algorithms, we set the weighting factor ω to $1/2$ in order to strike a perfect balance between memory and agility. T_{max} was set to 120 sec in both algorithms. For RSD, this meant that the timer $\mathscr{T}$ was reset to "120" sec at the beginning of each deferment period. As RSD depends also on the targeted $\overline{TUE}$, we ran it for 3 different values of $\overline{TUE}$: 1.2, 1.3 and 1.4, respectively. This gave us a sense of the sensitivity of RSD to $\overline{TUE}$.

In the experiment, we replayed the sequence of updates in each workload. During each replay, we recorded two measures on a per-deferment period basis: the length and resulting TUE of each sync deferment interval. For each workload, the results of both metrics were finally averaged to produce Fig. 3-4.

Results. As shown in Fig. 3, the TUE values of both ASD and RSD are very similar in the workloads $1-5$. These workloads are very intense, and show that both mechanisms are equally effective in the reduction of the network overhead in the face of frequent file modifications. For these workloads, RSD does not appear to be sensitive to the prespecified $\overline{TUE}$. This is a "good news", since it reduces the amount of potential "fine tuning" decisions to be made in order to optimize RSD's performance.

For workload 6, RSD is only comparable to ASD for the most loosen $\overline{TUE}$ value. The reason is that inter-update times in this workload often exceeded the 120 sec, and RSD ended up issuing a new sync operation with the cloud servers without attaining the targeted $\overline{TUE}$ in many occasions. However, such a behavior is desirable as confirmed in Fig. 4. Thanks to timer $\mathscr{T}$, RSD yielded synchronization times between 70X to 150X shorter than ASD, which incurred intolerably long sync delays.

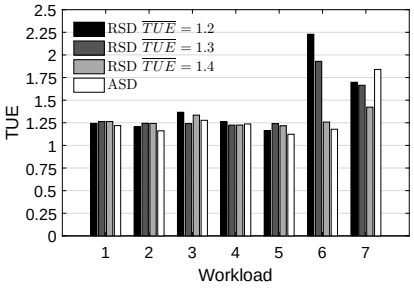

Fig. 3. Empirical TUE of ASD and RSD for different target $\overline{TUE}$ values.

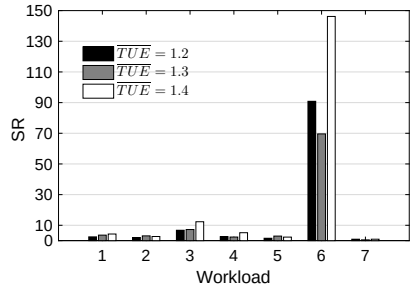

Fig. 4. Slowdown ratio (SR) of ASD relative to RSD for different $\overline{TUE}$s.

For workload 7, both algorithms reported bad TUE values. The reason behind this is that times between updates were very variable, alternating between long and short inter-update times, which caused a slow response of the EWMA controller in both algorithms. Even in this pessimistic setting, RSD was capable of decreasing a little bit the sync time as can be seen in Fig. 4.

For workloads $1-5$, RSD improved sync delays between 2X to 12X relative to ASD with equivalent TUE values, which undeniably demonstrates the superior performance of RSD.

V. CONCLUSION

Cloud storage services like Dropbox and Google Drive are becoming very popular these days. To optimize network traffic, these storage services rely on techniques like sync deferment. Literature so far has proven this technique to very useful in the face of frequent file modifications. However, there still exist some performance weaknesses on current implementations. To cope with them, this letter presents an innovative adaptive sync deferment algorithm, which is comparable to the current state of the art in terms of overhead, but as a distinguishing feature, it also optimizes file synchronization delays. Our experimental results report improvements between 2X to 12X in sync delay.

REFERENCES

[1] Z. Li *et al.*, "Towards network-level efficiency for cloud storage services," in *ACM IMC*, 2014, pp. 115–128.
[2] P. Garcia-Lopez *et al.*, "StackSync: Bringing Elasticity to Dropbox-like File Synchronization," in *Middleware*, 2014, pp. 49–60.
[3] Z. Li *et al.*, "Efficient batched synchronization in dropbox-like cloud storage services," in *Middleware*, 2013, pp. 307–327.
[4] R. Gracia-Tinedo *et al.*, "Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end," in *ACM IMC*, 2015, pp. 155–168.
[5] Z. Li, Z.-L. Zhang, and Y. Dai, "Coarse-grained cloud synchronization mechanism design may lead to severe traffic overuse," *Tsinghua Science and Technology*, vol. 18, no. 3, pp. 286–297, 2013.