Journal of Information Processing Vol.23 No.5 683–692 (Sep. 2015)

[DOI: 10.2197/ipsjjip.23.683]

Regular Paper

DiffHub: An Efficient Cloud Sync Technology based on Binary Diffs

Marat Zhanikeev[1,a)]

Received: January 6, 2015, Accepted: May 9, 2015

Abstract: Among several distinct kinds of syncing technology implemented in clouds today, none implement binary diffs for efficiency. Binary diffs are well established in literature and can be used to drastically reduce the bulk transferred over the network. Since most cloud technologies are distributed and depend on intensive internetworking, binary diffs can offer a considerable efficiency boost. This paper proposes the DiffHub method for cloud syncs. Its performance is analyzed separately on real filesystems and then on synthetic traces based on hotspot distributions. Results show that traffic bulk can be reduced by between 1 and 2 orders of magnitude, depending on conditions.

Keywords: diffhub, cloud sync, file sync, cloud drive, group drive, binary diff, bulk transfer, cloud migration

1. Introduction

The term **cloud sync** in this paper loosely defines the various cloud technologies that depend on intensive migration of resources over the network. This paper will specifically focus on *cloud drives* (Google Drive [26], Dropbox, etc.) which sync filesystems between a storage space in clouds and a local client. However, the same technology is applicable to migration of Virtual Machines (VMs), container-based apps (Docker [27]) and even traditional source code versioning (Github [28]).

Sync efficiency is mostly affected by the network where the related term is *bulk transfer* efficiency. There are several active research topics related specifically to cloud syncs. For the increased efficiency of VM migrations, *greyboxes* – reduced bulk of the image – are discussed [10], [11]. For BigData transfers, end-to-end circuit emulation is discussed as a method for achieving maximum throughput [16]. All discussions of *migration cost* show that network has the biggest effect [15], [17]. There are methods which take networking delay into consideration when building cloud-based APIs – like the example of over-the-network indexing in Ref. [19]. Finally, *mobile clouds* introduce a new concept of *group sync* where local wireless groups depend on intensive syncing of content across each other while the group as a whole also has to sync with the backend in the cloud. The related concepts of *MultiConnect* [20] and *GroupConnect* [21] are actively discussed in literature.

As a generic problem, distributed sync [3] and adaptive network-aware distributed sync [4] are the two actively discussed methods. Note that both these methods, as well as the P2P-based sync technology normally operate only one-way [4], while cloud sync, due to its design with the cloud backend mediating all the updates, can work in both directions.

Binary diff is a well-researched topic today. There is a current de-facto standard protocol/format for writing binary diff files – the VCDIFF based on RFC3284 [24]. There are also several command line tools, where VCDIFF-based *xdelta* [25] is arguably the most widespread today and can be found in many popular operating systems. Binary diff technology can be viewed as a more generic tool than the traditional *textual diff* – the latter is used in software development for version management. Binary diff is more generic because it can work with both text and binary files, while *textual diffs* work only with textual information and operate at the grain of *a line of text*. The currently popular *git* protocol and the Github [28] service are based on textual diffs. Further in this paper it is shown that binary diffs can often outperform traditional diffs.

Binary diffs are not used in clouds today. Specifically, none of the above technologies incorporate binary diffs, instead, relying on less efficient bulk reduction methods. An experiment further in this paper shows that Google Drive [26] transfers the entire file for each update (full sync). In Github, text files (source code, etc.) are processed with the traditional *diff* tool while binary files are transferred and replaced in full. Technologies related to BigData transfer, sharding in distributed filesystems of storage, and others, do not consider binary diffs, relying on other efficiency tricks instead [16]. Finally, binary diffs are not part of the two key cloud technologies – VM Migration [15] and container-based migration [27].

As a small taxonomy, the various diff technologies found in practice today can be classified into one of the following three kinds.

Textual Diff is the traditional kind found mostly in sourcecode version control (git and Github [28]). The command line *diff* tool is widespread and easy to use. The unit of such diffs is *one line of text*, where *end-of-line* symbol is used to cut text into lines.

[1] Department of Artificial Intelligence, Computer Science and Systems Engineering, Kyushu Institute of Technology, Iizuka 820–8502, Japan
[a)] maratishe@gmail.com

Filesystem Diff is a new concept that came to life recently via the Docker [27] service. Docker is a cloud-based hub for managing containers and container-based apps. Since a container is basically a size-optimized operating system, it is crucial for Docker to be efficient when implementing versioning of its containers. Today's Docker is entirely based on *git* – and even specifically the Github API, which means that filesystems in Docker are treated the same way as sourcecode trees are treated in Github. Efficiency is achieved through layers – Docker implements the *FROM* instruction which one can put into a *Dockerfile* in order to import a base container image. This helps Docker avoid duplication of huge – but otherwise static – base containers in the various apps that users build for themselves.

Binary Diff is the core concept used in this paper. Its main advantage is that it can replace both the above kinds. Text can be treated as a stream of bytes. Filesystems can be wrapped into a single compressed file (.zip in Github, .tar.gz or .tar.bz2 are also common) and subjected to binary diff. However, binary diffs can also potentially offer a higher level of flexibility. For example, each *FROM* instruction in Docker has to download the entire bulk of the specified base container. Using binary diffs, one can recreate a given container from a local library and transfer only the necessary diffs over the network.

Although not directly related to this paper, the concepts of *diff* and *patch* are the two sides of the same technology. The relation is simple – *diff* files can be used to *patch* an older file, thus resulting in a new (intended) file.

The idea for this paper was first born via the experiment in the next section which revealed that Google Drive uses full sync, that transfers the entire bulk of the file on each update. The same problem was later confirmed for Dropbox and several other publically available cloud drive services. The problem here is obvious – users have to wait a long time for a sync to propagate from one user to another. This was the exact problem that was experienced within a cross-univpersity research team (same as in the experiment below) that shared a cloud drive space.

The solution proposed in this paper is referred to as *DiffHub* and is a technology that uses cloud drive as an intermediary for diffs (rather than full files). Since diffs are much smaller in size, it takes much less time for updates to propagate to other clients (via the drive and) across the network. As a side feature, a drive that stores only diffs can handle 10–20 times more content, on average.

This paper also shows that besides the proposed DiffHub design, the technology is more flexible than the traditional design and can power interesting side technologies. For examples, initiation (first sync) of new clients can take hours with existing services. This paper shows that it can be sped up considerably using P2P file sharing [2]. If DiffHub is implemented at client-side – as is its current implementation – than the P2P option can easily become part of the new service.

This paper has the following structure. Section 2 introduces the experiment that involves a probe that helped reveal the fact that Google Drive uses full sync. Section 3 discussed related work. Section 4 describes the proposal and introduces the DiffHub technology. Section 5 explains how the experiment in Section 2 becomes a trace for analysis later in the paper. Section 6 discusses synthetic traces which are based on the hotspot distribution [16], [22] and are necessary for performance analysis over a wider range of conditions than those found in the real-life experiments. Section 7 explains the setup for analysis and Section 8 discusses the results. A conclusion is drawn in Section 9.

2. Justification via Experiment

As was mentioned before, the idea for this paper was born from the many reports on delayed syncs across users of a cross-university research team. Some members were working on the same dataset at roughly the same time and had to wait for a long time for syncs to propagate across the service.

It was decided to confirm the premise that Google Drive uses full sync – that is, sends the complete file of each update. **Figure 1** shows the topology used for the experiment. The topology describes a working Google Drive space shared across several members of a cross-university research team. The drive contained about 10 Gbytes of content. Note that Google Drive is classified as a *cloud sync* technology rather than the *distributed sync* discussed in related literature [4]. In cloud sync, all clients have slave copies of the content while Cloud Drive has the master copy. All the updates have to propagate to other users via the master copy. **Probe** is a newly added client to the topology and is put in charge of capturing *traffic* as well as *filesystem changes* in realtime.

The task of probing Google Drive only appears trivial. In reality, Google Drive protocol is not open source [26], while its client API is open. Without the full knowledge of the protocol, there is no clear way to separate metadata from sync traffic. Moreover, additional problems are introduced by the fact that metadata traffic is mixed with the bulk transfer of files as well as the fact that syncs are asynchronous with a changing time delay between uploads and downloads. Finally, the Probe needs to quickly detect changes in the filesystem so that captured packet traffic could be aligned with file syncs.

The following rules were established to deal with these difficulties. The Probe was passive – no outgoing, only arriving updates. To maximize the promptness of detecting filesystem changes, the Probe used *rsync* tool to find which files changed – the filesystem indexing used by the tool makes it the quickest known detector of changes in large filesystems [3]. The time when *rsync* detected a change was used as a reference time. The recent 40 s of traffic before the reference time would be collected and assumed as

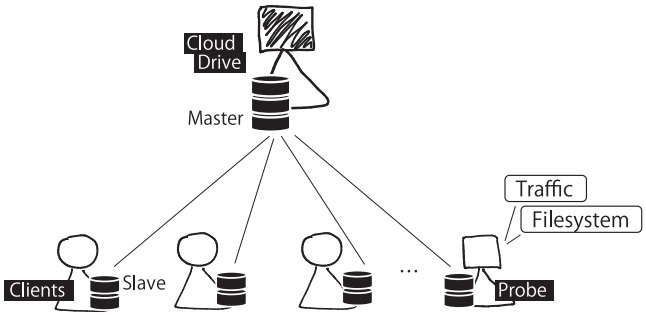

Fig. 1 Topology used for probing Google Drive syncs. All the probing was performed by the *Probe* client.

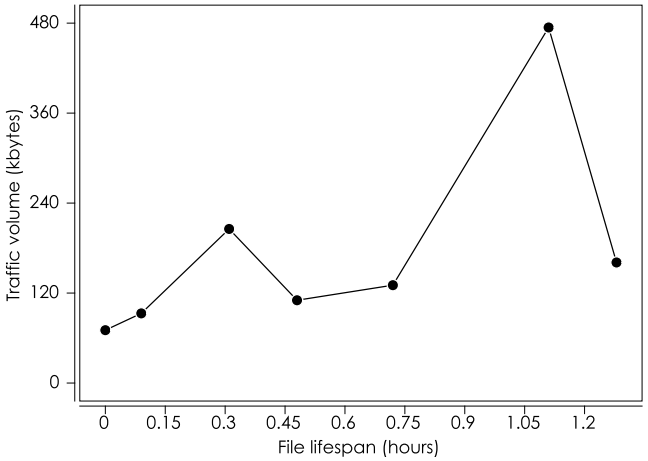

Fig. 2 An example lifespan of a file with the traffic received from Google Drive for each sync.

the volume of the sync. To purify the dataset, all episodes with multiple concurrent files were removed, thus, guaranteeing that each episode would contain only one updated file and the traffic corresponding to its sync.

Figure 2 shows an example lifespan of a file that grew from 100 kb to 250 kb. The figure shows that the dataset is noisy, with unexpected rises and falls of traffic volume. However, we can still see that updates experience a growing trend, thus, mirroring the growing of the filesize itself. This figure is the weak proof of the fact that Google Drive uses the full sync.

For a thorough analysis, a dataset for over 2,500 files – a mixture of text and binary files – was collected. Each update had at most several 10 kbytes of a diff (verified via binary diff). As a rule, the file always grew in size – this rule was followed naturally and there was no need to enforce it. The overall analysis was performed by observing cross-correlation (CCF: Cross-Correlation Function) between advancing *time* and *traffic volume*. Note that since files always grew in size, the same result would be achieved for CCF between filesize and traffic volume. Time was chosen as a *humanly readable* metric of performance – this paper started from the complaints about long waiting time.

Figure 3 has two plots that study CCF for individual files (upper) and filesize classes (lower).

The upper plot shows CCF for 35 randomly selected files. One can see that the majority of correlation is positive, that is, traffic volume grows along with the lifespan of a file. One can also see that there is a growing average trend in the plot – correlation is higher for larger files. Yet, the dataset is still very fuzzy to draw a final conclusion.

The lower plot attempts to separate noise from the valuable information. All the 2,500 files are aggregated into classes based on filesize (horizontal scale of the plot). The plot then shows the average CCF for each class. Here, we finally see the proof of direct correlation between filesize/lifespan and sync size. While for small filesize, the noise cannot be clearly separated from valuable information – metadata and various other traffic have heavy impact on the relatively small volume, for large size, the correlation is strongly around 30–40%. Note that this correlation cannot come from gradually increasing updates – as this paper shows

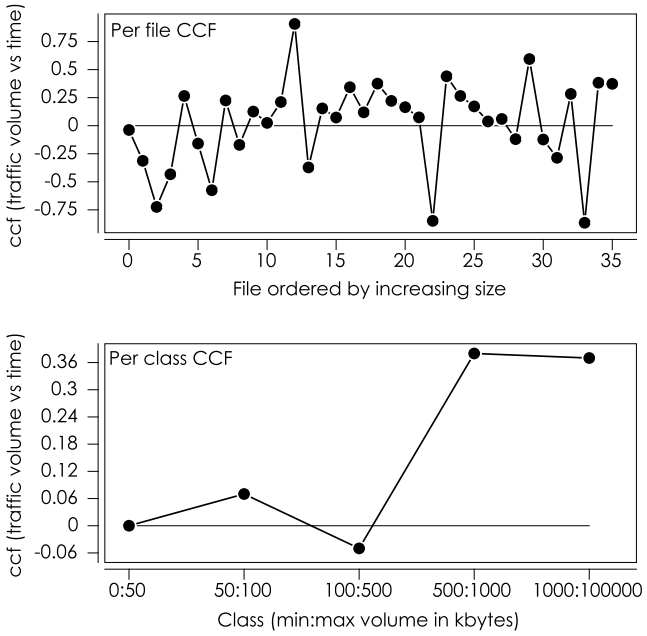

Fig. 3 CCF analysis of sync traffic volume versus filesize, separated into plots for individual files (*top*) and filesize classes (*bottom*).

further on, all the updates in the dataset are very small regardless of the total filesize. This means that the strong positive correlation can only come from the fact that Google Drive transfers the entire bulk of the file on each update.

Apart from the above numeric evidence, the findings were also supported by human observation. It always took longer for larger files to get synced across the service. Unfortunately, these reports were not confirmed numerically because it would require not only to install Probes at multiple clients, but also perform time synchronization across Probes. Along with having to deal with active clients (the above design uses passive measurement), such an effort would be considerably more difficult to implement in practice and would arguably create more complex (and noisy) datasets than the above.

3. Related Work

The binary diff technology is well established. VCDIFF (RFC3284 [24]) defines the abstract format and protocol for binary diff records. VCDIFF is used by the *xdelta* tool [25]. There are several other binary diff tools, but xdelta is the fastest and is more reliable. Xdelta can execute both diffing and patching parts of the process. Additionally, *xdelta* offers the feature of *merging multiple diffs* which is helpful for distributed asynchronous syncs because it can further reduce the volume of bulk transfer for delayed syncs. Note that DiffHub runs in asynchronous environments by definition and benefits from this feature – the hub can freely merge multiple diffs and update its other clients with a single binary diff even if its uploading client produced several diffs in succession.

VCDIFF format itself is very efficient because it uses compression internally. The algorithm for binary diffs uses several advanced elements of compression technology [1]. However, the main efficiency of binary diffs is in the algorithm for calculating the delta between two versions of the same file. *Xdelta* in this

respect is the leading product on the market. The documentation of the tool at Ref. [25] provides very good insight into the heuristics of the algorithm. Xdelta is in its 3rd major version and has undergone several major improvements. For example, one of the biggest changes in version 3 was the threshold between *detailed* versus *lightweight* heuristics (64 Mbytes but can be configured from command line). This switch allows the tool to avoid long computation times for extremely large files. Experiments in the previous section showed that *xdelta* can calculate diffs between two 3 Gbyte files in under a minute. It is interesting that the delta algorithm in *xdelta* uses a similar basic indexing technique to what *rsync* implements when calculating delta between two versions of filesystem states [3].

Note that the term *indexing efficiency* here can be divided into two types: indexing of filesystem changes versus changes in the content of individual files. Rsync is efficient only at the level of filesystem changes and does not index file content. VCDIFF is only concerned with binary diff between two versions of a single file. One way to view the proposal in this paper is to consider it an attempt to create a hybrid between the two – the proposed designs have both binary diff and filesystem components in one package.

Given this gap, there is already literature that adds binary diff to rsync [4] where the efficiency is helpful for distributed filesystems. However, note that technologies like *dsync* as well as the larger set of *peer syncing* technologies [4], [23] work in only one direction. The cloud sync design – where all the updates go through a central hub – is more robust in practice and can easily handle syncs in both directions.

There are several existing *efficiency tools* beyond the *dsync* above for file syncing in clouds. Majority of them are based on rsync but are not backed by academic papers (rsync itself is based on an academic paper [3]). It is still useful to introduce some of them to stress on the differences between popular tools today and the proposal in this paper. The efficiency of *gsync* [6] is related to very large files, where the tool implements the standard *rsync* algorithm for the entire local filesystem but is unique in making it possible to resume uploads and downloads for very large files. Actions can be resumed only sequentially and only if the file is not changed in the process. Otherwise, each new action involves the entire file without any efficiency. *S3sync* [7] is a utility for the S3 cloud storage run by Amazon, the only efficiency here is in that it implements the *rsync* algorithm and offers the simplified *sync* utility to the user instead of the raw per-file actions provided by the basic S3 command line tool. There are also examples when cloud storage services offer *rsync* interfaces for syncing – see the example of the popular *iBackup* service [8]. Note that *rsync-like* interfaces can already be considered as improved efficiency for cloud storage where, including the case of cloud drives (Google Drive, Dropbox, etc.) considered further in this paper, files are stored in the cloud individually rather than mirroring the local filesystem tree of the client.

Zsync [9] is the only popular tool that advertises the use of *binary diffs* for efficiency. *Zsync* is divided into client and server sides. The client implements the basic *rsync* algorithm to find out which files to sync. The server provides *.zsync* binary delta files for each actual file which each client can use to patch its own old version. In this case, only the delta file is downloaded.

Both *dsync* (*rdiff* [5] follows the same concept) and *zsync* use binary diff as part of their functionality. However, in both cases, binary diffs are always applied only to individual files. The proposal in this paper is original in that it describes a platform that is *centered* around binary diffs. The platform contains crucial components like *multi-user access* and *version control*, which are missing in *dsync* and *zsync*. *Dsync* is limited to the case of one-to-many syncing where the single source always has the most recent copy. *Zsync* can only support multiple clients if the server has the most recent copy and all clients have the same older copy. Neither tool supports version control. This classifies them as *backup* tools – a subclass of the larger *syncing* technology considered in this paper. The rest of this section shows several technologies which can benefit from the proposal while being unable to apply *dsync* or *zsync* due to the lack of one of the two above components.

VM migration suffers extensively from the cost of migration. While the traditional *VM placement* technology in clouds does not consider migration cost [13], [14], modern methods are cost-aware [12]. It is widely accepted that the cost of migration is mostly the bulk transfer over the network [17].

The *greybox* problem here is defined as a method that reduces the bulk of a migrating VM [10]. Unique – and therefore less backwards-compatible – designs are used as well, for example, as APIs that separate OS from apps and data and transfer only the necessary parts over the network [11]. This paper shows that binary diffs on VM images can be very efficient as well. Note that in this case VM images are treated as blackboxes – that is, there is no need to know what is inside a VM image. This makes the binary diff approach more flexible than the above API.

Container-based migration suffers from similar problems. Docker [27] being the popular container hosting service today is an example of a *filesystem diff* technology. A given user's container is built in layers. Lower/base layers are mostly static and can be imported as a base running environment for a given container-based app using the *FROM* instruction in Docker. The layers make it possible for Docker to minimize the impact from having to deal with binary files (kernel, etc.) in OS images. Docker assumes that all the binary files are updated very infrequently. This means that Docker can exist without the binary diff and use the simpler textual diff technology. Specifically, current Docker is based on *git* tools and more specifically on *Github* APIs. This paper shows that binary diff can be very efficient for filesystem images where the entire container can be handled as a single compressed file.

Cloud syncs go beyond file syncing. Recently, syncing technology has become important for *mobile clouds* where syncs are separated into local (in-group) syncing and group-to-cloud syncing [21]. The more general concept of *GroupConnect* is proposed in Ref. [20]. There are many practical applications for GroupConnent [21] but all rely on more than one kind of syncs within the same technology.

While the concept of group sync is natural to wireless environments, the same concept can be applied to wired communities.

For example, the *Group Drive* technology is a form of P2P syncing aiming at realtime content handling by groups in classes [23]. Note that all these examples depend on full sync and can therefore benefit from the efficiency introduced by binary diffs.

Repeating an earlier statement, syncing performance is mostly affected by end-to-end networking. Therefore, any technology that improves networking, by extension, improves the performance of syncs. All the above technologies focus on reducing the bulk itself. There is also research that tries to improve end-to-end throughput. For example, end-to-end circuits are discussed as a special case of efficient networking for BigData, VM images and other very large bulks found in clouds [16]. There is also literature that does not improve but takes into consideration network delay – the example technology here is the over-the-network indexing in Ref. [19]. Note that most cloud technologies natively operate in over-the-network mode which means that the findings apply to all cloud syncs. The same premise applies here – reduction of the bulk itself via binary diffing can boost the performance of all such technologies.

Although not directly related to the scope of this paper, a recent cloud technology discussed in literature is *multistream content aggregation* – where content is aggregated in realtime from multiple peers in P2P networks [18] or cloud-based services [2]. While the technology has P2P features, there are several features that make it a distinct technology. The content aggregation technology is related to this paper in the part of so-called *initiation* – where the entire contents of a cloud drive has to be synced with a newly added client. Practice today shows that such a sync can take hours. This paper shows that this time can be minimized by using content aggregation as defined in Ref. [2].

4. Proposal: the DiffHub

This section formulates the concept of DiffHub as well as its implementation designs and features.

Figure 4 shows 3 distinct types of cloud syncs. All are described as procedures undertaken when a file is updated and the update has to propagate from the origin (User A) to Cloud Drive, and finally to User B on the other side.

The **Traditional** design follows the trends discovered in the experiment above. New file is uploaded to Cloud Drive in full, where it replaces the old file (versioning is discussed further in text). Client at User B performs continuous polling on status and promptly finds out about the updated file. The new file is then downloaded and used to replace the old version (no versioning here, the old version is deleted).

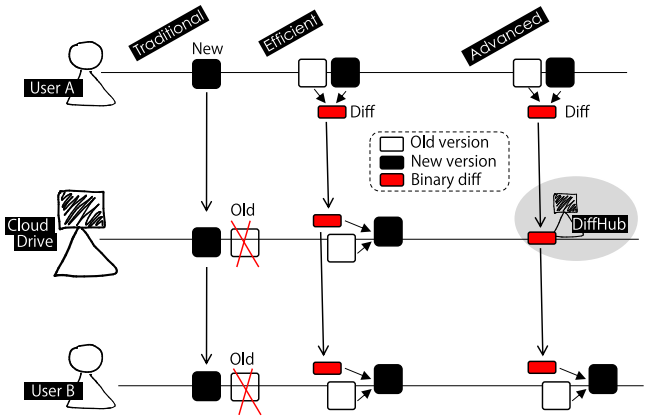

Fig. 4 Taxonomy of types of cloud syncs, with the proposed DiffHub marked as the *Advanced* design.

The **Efficient** design showcases the immediate efficiency from using binary diff. The diff is created by the client at User A as a product of comparing old and new versions of the same file. The relatively small diff file is then uploaded to Cloud Drive, where it is patched on the old version to create the new version. User B in turn downloads the diff from Cloud Drive and performs the same set of actions to create the new file. To support version discrepancies across multiple clients in async environments, diffs for several versions (diff versioning) can be implemented, where *xdelta* tool can merge several diffs into one, potentially creating the final diff of a smaller size than the sum of its predecessors.

The **Advanced** design is the main objective of this paper. Cloud Drive here does not retain entire files but handles only diffs. Hence the name *DiffHub* applied to the design. Note that such a design would require a complete rewrite of the software at Cloud Drive. However, in practice this drastic upgrade can be avoided by emulating a DiffHub. The current implementation of DiffHub simply uses a discipline of storing only diffs in Google Drive while keeping the files themselves in a location outside of Cloud Drive's local filesystem. Cloud Drive in this case follows its normal operation routine while the core logic of the DiffHub is implemented at client side.

The unique differences between Advanced and the Traditional/Efficient designs are as follows. The Advanced design (Diffhub) is unique compared to both other designs in that it stores only diff files, while the other two also have to store the main bulk. The Advanced design is partially unique compared to Traditional design in that only diffs are used for communication between clients and Cloud Drive – the uniqueness is partial becuase the same feature is offered by the Efficient design.

Based on the above description, DiffHub (Advanced design) has both pros and cons. First, the fact that syncs in both Efficient and Advanced models are smaller (and therefore faster), it is the first obvious pro of the technology. Another pro is the storage space, which is more efficient in the Efficient design (updates are stored as diffs) and reaches maximum efficiency in the Advanced design.

The main disadvantage for the Advanced model is the fact that it cannot support *initiation* of new clients in its default form. The solution for this problem is offered later in this section. Another disadvantage is hidden in version control logic. Resolution of collisions has always existed as a fundamental problem [4]. In practice (including the practice with the team from the above experiment), it is almost always the case that User A does not know what User B did, which means that a collision cannot be resolved by either user without direct communication between the two. The discussion of versioning is left out of scope of this proposal and will be revisited in future publications. If versioning is crucial for a team's operation, then the Efficient design is advised – it is still more efficient than the Traditional design while implementing the same versioning logic.

Figure 5 shows the solution to the initiation problem and is

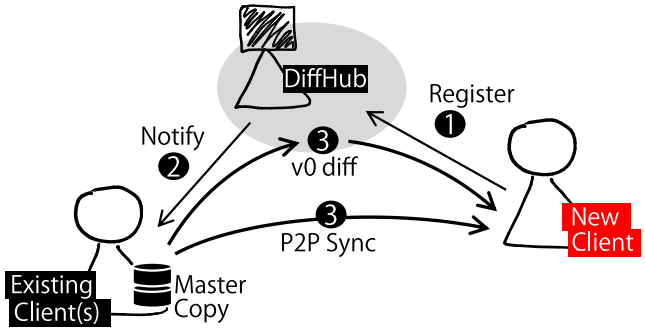

Fig. 5 Resolving the initialization problem for newly added clients. Two separate methods are proposed in Step 3.

specific to the Advanced (DiffHub) design. Note that this design is drastically different from the other two in that all clients have the master copies of the content – thus are more likely to be classified as a P2P sync technology [4]. The following steps are undertaken for each newly added client:

(1) A new client registers with DiffHub;

(2) DiffHub notifies (or, alternatively reads status updates for) all existing clients. DiffHub does not store the files themselves, so, it depends on other clients for supplying the new client with current versions of the files.

(3) This step can be implemented in two distinct ways. One is via the *v0diff* (reads version zero diff) function which is where the binary diff file contains the entire bulk of the file. The other method is to implement P2P sync which is where the new client aggregates the entire initial state using multi-stream aggregation [2].

Clearly, the second method of Step 3 is much faster but requires additional software for each client. If clients are in a local network then GroupSync can be used [4], [22]. If existing clients are remote then P2P [2] or cloud-based multisource [2] are the available options. The current version of DiffHub implements the relatively simple *v0diff* (using the VCDIFF notation) while future publications on the topic will look into P2P options.

5. Experimental Traces

This section extends Section 2 by discussing additional details of the experiments as well as forming a trace for further analysis. The entire trace (dataset, etc.) contains over 6 months of operation of a cross-university research team. In this period, diffs for over 2,500 files have been collected. Majority of them have been synced over Google Drive, but some were too big for the space and were handled outside. Regardless, a discipline was enforced which collected diffs for large files and added them to the data collected automatically by the Probe.

Table 1 shows some of the common filetypes encountered in the trace. The filetypes were selected manually but the filesizes were picked as the most frequent (rounded) number found in the trace. Diffsize was defined as the average number across all the selected syncs.

The following features were discovered based on Table 1. Source code (programming) generates the smallest diffs, even when taken relative to the total filesize. This is mostly the fault of the auto-sync feature which saves current version of the file

Table 1 The table of several kinds of binary diffs that are useful in clouds and their performance.

Application or Filetype	Filesize	Diffsize	Diff %
SC: Source Code (1min auto-save)	~40kb	0.2kb	0.5%
PPT: Power Point	100kb	10kb	10%
DOC: Word Document	60kb	5kb	~10%
PDF: PDF Annotation	100kb	2kb	2%
ZIP: Archive (Github commits/exports)	2Mb	2kb	0.01%
VM1: VM Images (xva, small)	800Mb	150Mb	20%
VM2: VM Images (xva, large)	2Gb	200Mb	10%

automatically if the content has been edited. Yet, manual saves were found to be at most 2–3 times bigger (programmers like to save frequently) which is not a major difference.

Office application diffs (per save) are at about 10% of filesize, regardless of filesize and application type. PDF annotation (adding comments to PDF files) is very small – about 2%. Zip files of source code (Github commits were downloaded and tested) are also very small. Note that this has direct relation to Docker which handles its containers as Github repositories.

VMs are separated into 1 Gb and 2 Gb classes where the former diffs are at 20%, and the latter at about 10% of the total filesize of the image. Each time new VM images were created after major software of environment updates or installations of new software or applications. Note that diffs are closer to each other in filesize than the images themselves – here the premise is that diffs do not grow much in size regardless of the total filesize. Of course, this premise only holds on condition that we are comparing incrementally updated images – two unrelated images would generate huge diff files. Also note that these diffs can compete with the success of greyboxes [11] without the need to use non-standard APIs to separate apps and data from OS parts of the image.

Figure 6 is generated from the same experimental data as was presented in Section 2, but this time focuses on diffs. The plots are separated into binary diffs only (*top*) and comparison between three kinds of diffs (*bottom*).

The top plot shows 35 randomly selected files as vertical pairs of filesize versus diff size. We can immediately see that diff size does not correlate with file size – relatively large files (vertical scale is in log) can have diffs of the same size as that for small files. As was mentioned before, most diffs in the dataset are in the range between 100 s of bytes and 10 s of kbytes.

The bottom plot compares the performance of *text* versus *compressed text* (zip) versus *binary* diffs. The samples are ordered by the decreasing size of binary diff with the two other kinds plotted on top. Each vertical group corresponds to the same (text) file during the same sync. We can see that text diff performs better on average than binary diff, with occasional lapses where text diff is much bigger. The plot also advises against using compressed textual diffs, which only smoothens both positive and negative advantages of the purely textual diffs. The problem here is that compression does not perform well on small snippets of text. The performance should be better if diff files are bigger.

The above dataset is converted to the trace as is. This means that the actual file and diff sizes are used for analysis further in this paper.

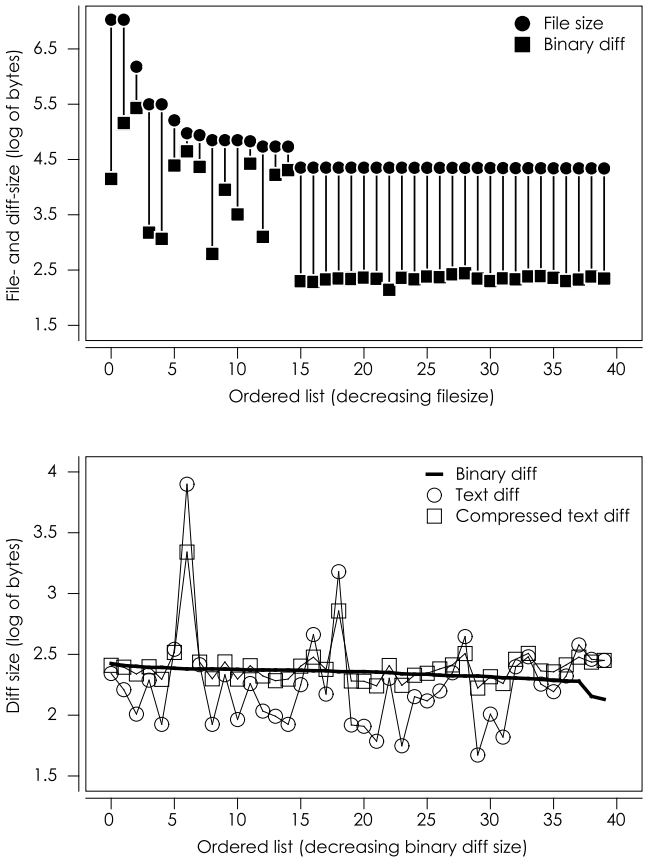

Fig. 6 A randomly selected subset of real measurements showing only binary diffs (*top*) and comparison between three diff formats (*bottom*). Logs on vertical scales are base 10.

6. Synthetic Traces

The real traces above raise an obvious problem. While the trace comes from real measurements, it only describes a relatively small subset of possible usage patterns. The synthetic trace described in this section rectifies this problem by introducing a method for generating realistic but synthetic traces.

The best basis for synthesis is the so-called *hotspot distribution* – a recently developed method for describing systems with multiple parallel random processes of which only a small subset experiences extreme behavior. There are many examples of hotspot processes: viral (very popular) videos on a Content Delivery Network (CDN), BigData versus ordinary bulk [16], and filesize distribution in this paper. This paper uses the hotspot distribution to model *filesize*.

Full description of the synthesis method and the involved statistical processes can be found in Ref. [22]. The standard method is a process executed in discrete time but a simplified version has also been proposed [16]. The simplified version is static and produces 4 sets of *normal*, *popular* (pop), *hot* and *Flash Crowd* (flash) items. The normal in the simplified version is all set to 1 and is therefore not interesting. The *pop* set is part of the emulated filesystem but describes the files that never get updated and is therefore also outside of the current scope. Finally, *hot* and *flash* sets are the two key sets that define the initial and final filesizes for each item, respectively. Note that this application of the hotspot distribution is similar to BigData modeling in Ref. [16].

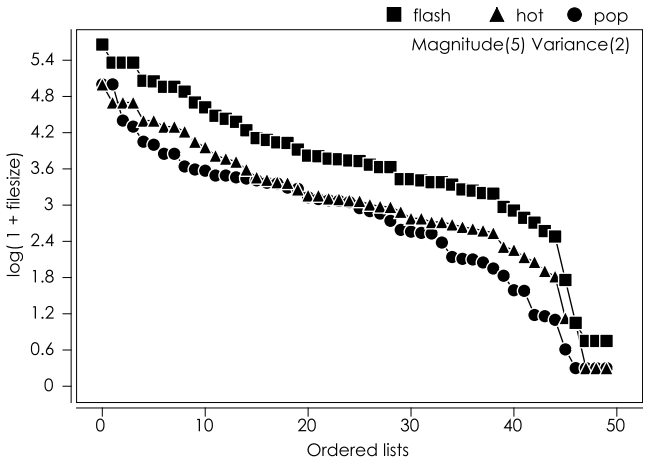

Fig. 7 A randomly selected hotspot distribution. Vertical scale is log base 10.

Setting the time component aside, the hotspot distribution is modeled using the following tuple:

$$\langle n, m, k, a_d, a_m, a_v \rangle, \tag{1}$$

where n, m, and k are *normal*, *pop*, and *hot* item counts (*flash* count is same as hot), a_d configures the Stick-Breaking (SB) random process, and a_m and a_v define magnitude and variance of the Dirichlet distribution that configures independent random streams for each hot item. Further reducing complexity by fixing $a_d = 0.5$, we have the two main parameters a_m and a_v that can generate a wide variety of distributions.

Figure 7 shows a randomly selected hotspot distribution with magnitude set to $a_m = 5$ and variance to $a_v = 2$. For visual clarity, all sets are plotted as ordered distributions which means that vertical alignment does not correspond to the same item. In the synthetic trace, each item has an independent set of *hot* and *flash* values. In the plot we can see that *hot* filesize is slightly larger (even in initial state) than of pop items. In the flash set, filesize of hot items experiences a major surplus.

While hotspot distributions are generated using a complex process, the two above parameters can help understand the nature of the configuration control they offer. Magnitude controls the upper margin – not a fixed bound but the magnitude of the spikes. Variance controls the variety of magnitudes across hot items – here the counterintuitive feature is that higher variance can actually cause lower average by generating too many extreme (low and high) fluctuations.

As was mentioned earlier, values in the above hotspot distributions correspond to filesize. To adjust the values to a realistic range of filesize, all the values are multiplied by 10, thus, creating large files in the range of several Mbytes.

7. Analysis Setup

The target of the analysis in this paper is to compare performance between the proposed DiffHub and traditional Cloud Drives. The main performance metric remains the same throughout this entire paper – waiting time for individual syncs. *Wating time* is defined as the time spent by the client waiting for one sync to complete. For clarity, analysis further in this paper focuses on comparing between *Traditional* and *Diffhub* designs.

The *Efficient* design can be considered as a hybrid in which on one hand waiting time is the same as that of the *DiffHub* because only diffs are transfered, and on the other hand the storage space at server side is comparable to that of the *Traditional* design. *Storage space* is not considered in analysis becuase its performance mirrors that of traffic exchange. In fact, the relation between the two metrics is strongly proportional becuase large storage space can be filled by high-volume traffic exchange, and vice versa.

Waiting time depends directly on end-to-end throughput. To simplify analysis, network throughput is fixed at 5 Mbps for all syncs (real and synthetic, v0diffs and P2P, all filesizes). In practice, the experimental data shows that most of the throughputs (3σ band) fall within the 2–4 Mbps range for the specific case of Google Drive, so 5 Mbps is an optimistic value. Waiting time is calculated simply as filesize (multiplied by 8 for bits) divided by throughput (in bps). The fixed throughput helps focus on the effect from other parameters. For example, because of the fixed throughput, waiting time is directly related to the traffic volume produced by each method.

The real trace is handled in the following form. The 2,500 files in the trace form about 50% of the entire filesystem in cloud drive (the above experiment). Most of these files were generated automatically by running simulators, or processing data in research projects but some (like office) were handled manually. Files are strictly growing in size. The only parameter in analysis is the *ratio* of files, where a given value is used to selected a random subset of the real files in the trace. The range of the analysis is therefore between 5% and 50% of the entire filesystem that existed during the experiment (Sections 2 and 5). Actual diffs for each selected file are then replayed as they were recorded in the trace by the Probe.

The synthetic trace is handled in the following form. First, it offers a richer set of conditions and therefore has more parameters. One parameter is a set of combinations of the *magnitude* and *variance* parameters used to generate the hotspot distribution that defines all the filesizes. The other parameter is the relative size of *diffs*, which can be between 5% and 15% of the initial filesize of a hot item. Each file then experiences a set of increments (of each individual diff size) until filesize reaches the value in the *flash* set. Each file is replayed separately with waiting time readings collected for each diffs. All hotspot distributions have 100 items in its sets.

Both real and synthetic simulations were conducted until 100 samples were collected for each combination of the respective parameters. These 100 samples are viewed as averages and variance (3σ) band when discussing the results.

8. Analysis Results

This section discusses analysis results. The analysis itself follows the setup expalained in the previous section. Same as previously in this paper, the discussion of performance is cleanly separated into *real* and *synthetic* parts. The merit/demerit of the real part is that the performance is derivied from real-life data but fails to cover a sufficiently wide range of configurations. The merit/demerit of the syntheric part is the opposite – the widest possible range of configurations is tested but it is hard to connect each configuration to a real situation. Both parts should be viewed together when forming the overall judgement about the *DiffHub* technology and its advantages over the *Traditional* design.

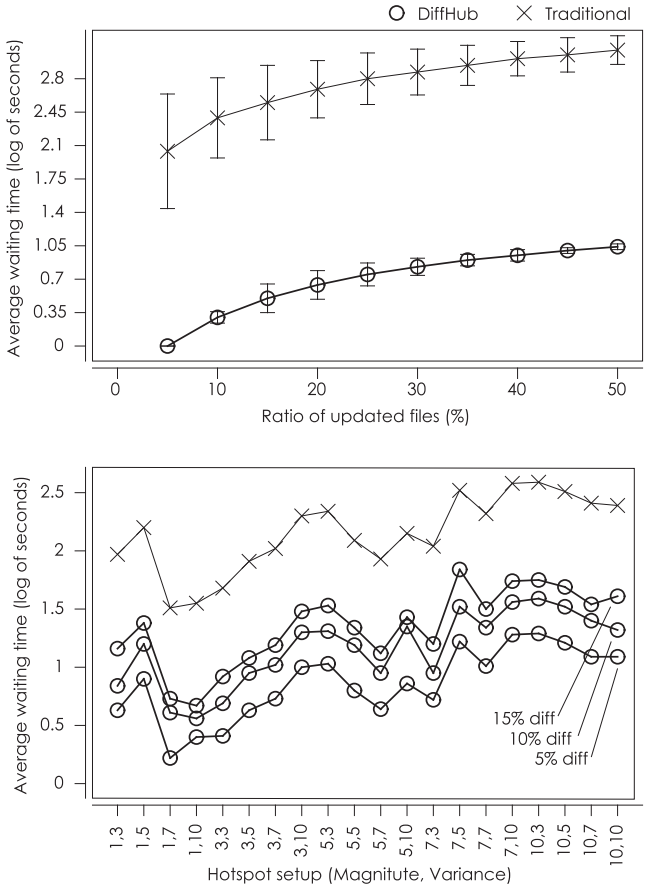

Fig. 8 Real (*top*) versus synthetic (*bottom*) results. 3σ variance bands are shown for each bullet. Vertical scales use log of base 10.

Figure 8 shows the performance for real (upper) and synthetic (lower) traces. As was explained above, the viewpoints for each plot are different. The upper plot views waiting time against the ratio of updated files while the lower plot sees how a given hotspot distribution affects performance in both the compared designs.

The real performance is as follows. Most real diffs (upper plot) are small, on average *2.5 orders of magnitude* smaller than the respective filesize, meaning that the waiting time can be reduced by over 2 orders of magnitude. Note that the small diffs are possibly due to the specifics of this particular trace where the diffs were produced from autosaves of source code or relatively frequent saves in office applications. Performance of binary diff in respect to small versus large and text versus binary files was discusses earlier in this section based on the experimental data. This same effect is reflected here – majority of the diffs are small, with only a few really large updates (see the above section for specific numbers).

Each bullet in the upper plot in Fig. 8 shows the 3σ variance band which makes it possible to judge how scattered is each aggregation of values. Indirectly, large variance can be linked to *low reliability* of a given level of performance.

Let us review the upper plot in the left-to-right order. When the number (ratio) of files is small, we can see that syncs can propa-

gate very quickly – and reliably (small variance) in the DiffHub while traditional syncs can cause a wide range of waiting times (high variance). This reveals the smoothing effect of binary diffs where diffs can be small even for large files. In fact, diff size but definition is not related to the total file size but is derivied only from the size of the updated portion. When only few files are updated, performance of the Traditional design is widely scattered because large files are rare and may not be found in every batch. This does not affect DiffHub performance because diffs are small even for most large files.

With increasing ratio (a richer mixture of files), the variance decreases while the average waiting time increases. Interestingly, the variance for the DiffHub model is highest for the mid-range of the ratio (around 15–25% of the total filesystem). This is because large diffs in the trace are even rarer that large files and reveal themselves at a higher ratio. This artifact has a practical value. Traditional design can expect wide fluctuations in performance even when the ratio of files is small, while for the DiffHub design such a point occurs at much higher ratios.

An interesting practical reading of the upper plot in Fig. 8 is that it takes, on average, under 10 s for the DiffHub to transfer all the diffs, while the Traditional method would take 1,000 s to complete its sync.

Performance based on the synthetic traces is much richer (bottom plot of Fig. 8) due to the wide range of hotspot distributions. The hotspot configuration is listed on the horizontal scale in form of pairs of values. The plot has a slightly cyclic view because hotspot setup changes in two loops – magnitude forms the outer loop and variance the inner. Note that changing diff ratio has no effect on the traditional model which is why it is represented as the single curve.

The best performance is registered at the $x = 1,7$ point. This condition is the combination of low magnitude and high variance in filesize. The worst performance is at $x = 7,5$ point where both magnitude and variance are in mid-range. This later set contains a small number of very large files (not much variance between them) which cause larger syncs on average. The entire set for magnitude set to 10 shows reliably poor performance (but the same relative performance) simply because all such distributions have very large files.

Note that hotspot distributions cannot be classified using simple characteristics like *heavy* or *light* easily. Even with the two parameters in the figure, the horizontal scale is not presented in an increasing/decreasing or even cyclic form. Instead, the horizontal scale should be viewed as a range of conditions to which each design is subjected.

The overall trends in the synthetic performance are as follows. Both magnitude and variance affect performance. However, while growing magnitude reliably correlates to longer waiting time, higher variance can both decrease and increase waiting time due to random combinations of filesize in the respective sets. For example, values 5 and 7 for variance often cause opposing fluctuation in performance.

The practical reading from the bottom plot of Fig. 8 is that it takes between 50 s and 500 s for the Traditional method to complete it syncs while the DiffHub can accomplish the same job in between 5 s and 50 s. The difference is about one order of magnitude for the entire range of conditions. Note that this is an optimistic outcome given that the real traces above showed the gap of 2 orders of magnitude. Even if these two outcomes can be considered as opposite extremes, the DiffHub can be expected to peform between 1 and 2 orders of magnitude better in any setting.

Note that, while the above plots show that the trends in waiting time performance heavily correlate between the traditional and DiffHub models, the relation is not direct and is mostly smoothed out by the log scales in the plots. Traditional model causes multiple transfers of the entire bulk of the filesize while the DiffHub model exchanges much smaller diffs.

The main target of the analysis – visualize performance bounds for the waiting time – is fully achieved. Analysis based on real traces shows that waiting time under DiffHub can be reduced from several hundreds of seconds to under 10 s, on average. Synthetic traces show that in the extreme cases, DiffHub can reduce waiting time on average by close to one order of magnitude and depends weakly on the ratio of diffs to the size of individual files.

9. Conclusion

This paper proposed a cloud sync technology called DiffHub, the core component of which is binary diff. The immediate practical problem solved by DiffHub is the long waiting time for syncs across clients in clouds drives – where the experiments in this paper were conducted on Google Drive. The waiting time was reduced by 1–2 orders of magnitude, depending on conditions.

This paper performed analysis in two kinds of conditions. Real conditions were recreated based on experimental data and showed that waiting time can be reduced by 2 orders of magnitude on average. Synthetic traces helped create a much more diverse set of conditions and reduced waiting time by between 1.5 and 2 orders of magnitude.

The proposed technology was shown to exist in two practical forms. A minor upgrade can be facilitated simply by exchanging diffs between clients and cloud drive while retaining the old technology in the remaining design. While the same advantages are offered by this model in terms of waiting time, the advantage here is that version control and initiation of new clients can be done in the traditional way.

The most advanced design referred to as DiffHub lacks compatibility both in version control and initiation of new clients – both due to the fact that DiffHub does not store the entire files. However, this paper proposed two solutions to the initiation problem. The first solution is to use version zero diffs (v0diff, the entire file is a diff) which is relatively easy to implement in practice in a way that is compatible with the binary diff toolset. The other method is for the new client to aggregate the initial state from multiple existing clients using P2P methods, where recent literature offers new P2P methods for realtime parallel content aggregation. Although this method requires a major effort to implement in practice, it offers the most benefits because waiting time for the initiation sync would be substantially reduced.

The current version of DiffHub implements the simple v0diff method. The current version is also an emulation where the traditional cloud drive (Google Drive) is used as is while all the

DiffHub functionality is implemented by clients. The proposed design remains unchanged – cloud drive is used only to store the diffs of files while the files themselves exist only at clients. Note that this immediately increases the capacity of the cloud drive – based on the numbers in this paper by at least the order of magnitude.

Future work will implement the DiffHub as a full cloud service, thus, making it possible for clients to switch to a new service rather than to have a hybrid DiffHub + cloud drive installations. The P2P method of initiation will be looked into as a means of providing fast syncs both for file upgrades as well as for initiation. This means that this advanced form of DiffHub would offer excellent waiting time performance for the entire set of its functions.

References

[1] Nelson, M.: *The Data Compression Book*, Henry Holt and Co., New York, USA (1991).
[2] Zhanikeev, M.: Multi-Source Stream Aggregation in the Cloud, *Advanced Content Delivery and Streaming in the Cloud*, Wiley (2014).
[3] Tridgell, A.: Efficient Algorithms for Sorting and Synchronization, Doctor Dissertation, Australian National University (2001).
[4] Pucha, H., Kaminsky, M., Andersen, D. and Kozuch, M.: Adaptive File Transfers for Diverse Environments, *USENIX Annual Technical Conference*, pp.157–171 (Aug. 2008).
[5] rdiff: signature-based file differences for rsync (online), available from ⟨http://linux.die.net/man/1/rdiff⟩ (Retrieved Mar. 2015).
[6] gsync tool syncing very large files wtih Google Drive (online), available from ⟨https://code.google.com/p/gsync/⟩ (Retrieved Mar. 2015).
[7] s3sync tool for syncing files on S3 (online), available from ⟨https://github.com/clarete/s3sync⟩ (Retrieved Mar. 2015).
[8] rsync in context of iBackup cloud service (online), available from ⟨https://www.ibackup.com⟩ (Retrieved Mar. 2015).
[9] zsync file transfer software (online), available from ⟨http://zsync.moria.org.uk/⟩ (Retrieved Mar. 2015).
[10] Wood, T., Shenoy, P., Venkataramani, A. and Yousif, M.: Black-Box and Gray-Box Strategies for Virtual Machine Migration, 4th USENIX Symp. on Networked Systems Design and Implementation, pp.229–242 (2007).
[11] Antonio, C., Tusa, F., Villari, M. and Puliofito, A.: Improving Virtual Machine Migration in Federated Cloud Environments, *Second International Conference on Evolving Internet*, pp.61–67 (Mar. 2010).
[12] Andreolini, M., Casolari, S., Colajanni, M. and Messori, M.: Dynamic Load Management of Virtual Machines in a Cloud Architecture, *ICST CLOUDCOMP*, pp.201–214 (2009).
[13] Dhiman, G., Marchetti, G. and Rosing, T.: vGreen: A System for Energy Efficient Computing in Virtualized Environments, *14th ACM/IEEE International Symposium on Low Power Electronics and Design*, pp.243–248 (2009).
[14] Xu, J. and Fortes, J.: Multi-objective Virtual Machine Placement in Virtualized Data Center Environments, *IEEE/ACM International Conference on Green Computing and Communications* (*GreenCom*) *jointly with Conference on Cyber, Physical and Social Computing* (*CPSCom*), pp.179–188 (Dec. 2010).
[15] Zhanikeev, M.: Optimizing Virtual Machine Migration for Energy-Efficient Clouds, *IEICE Trans. Communications*, Vol.E97-B, No.2, pp.450–458 (Feb. 2014).
[16] Zhanikeev, M.: Circuit Emulation for Big Data Transfers in Clouds, *Networking for Big Data*, pp.359–393, CRC (2015).
[17] Voorsluys, W., Broberg, J., Venugopal, S., Buyya, R.: Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation, *CloudCom*, pp.254–265 (2009).
[18] Zhanikeev, M.: A Method for Extremely Scalable and Low Demand Live P2P Streaming based on Variable Bitrate, *1st International Symposium on Computing and Networking* (*CANDAR, former ICNC*) (Dec. 2013).
[19] Zhanikeev, M.: A New Practical Design for Browsable Over-the-Network Indexing, *International Conference on Information Science, Electronics and Electrical Engineering* (*ISEEE*) (Apr. 2014).
[20] Zhanikeev, M.: Virtual Wireless User: A Practical Design for Parallel MultiConnect Using WiFi Direct in Group Communication, *10th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services* (*MobiQuitous*) (Dec. 2013).
[21] Zhanikeev, M.: Opportunistic Multiconnect with P2P WiFi and Cellular Providers, *Advances in Mobile Computing and Communications: 4G and Beyond*, CRC (in press) (2015).
[22] Zhanikeev, M. and Tanaka, Y.: Popularity-Based Modeling of Flash Events in Synthetic Packet Traces, *IEICE Technical Report on Communication Quality*, Vol.112, No.288, pp.1–6 (Nov. 2012).
[23] Zhanikeev, M. and Koide, H.: YALMS: A Group Drive API for Cloud-Based Classrooms, *IEICE Technical Report on Information Network* (*IN*), Vol.113, No.303, pp.19–22 (Nov. 2013).
[24] The VCDIFF Generic Differencing and Compression Data Format, RFC3284 (June 2002).
[25] XDelta: A tool for binary diffs (online), available from ⟨http://xdelta.org⟩ (Retrieved Dec. 2014).
[26] Google Drive (online), available from ⟨http://drive.google.com⟩ (Retrieved Dec. 2014).
[27] Docker platform (online), available from ⟨https://www.docker.com⟩ (Retrieved Dec. 2014).
[28] GitHub (online), available from ⟨http://github.com⟩ (Retrieved Dec. 2014).

Marat Zhanikeev received M.S. and Ph.D. in Global Information and Telecommunications Studies from Waseda University in Tokyo, Japan, in 2003 and 2007, respectively. His research interests include network measurement, network monitoring, and network management, but also extend to practical applications related to these topics as well as non-traditional applications of information technology in general. He is presently an Associate Professor at Kyushu Institute of Technology (Kyutech), and is a Regular Member of IPSJ.