2016 IEEE 17th International Symposium on High Assurance Systems Engineering

D^2PS: a Dependable Data Provisioning Service in Multi-Tenant Cloud Environment

Renyu Yang[†], Tianyu Wo[†], Chunming Hu[†], Jie Xu[§†], Mingming Zhang[†]
[†]School of Computer Science and Engineering, Beihang University, Beijing, China
[§]School of Computing, University of Leeds, Leeds, UK
Email: {yangry, woty, hucm, zhangmm}@act.buaa.edu.cn; j.xu@leeds.ac.uk

***Abstract*—Software as a Service (SaaS) is a software delivery and business model widely used by Cloud computing. Instead of purchasing and maintaining a software suite permanently, customers only need to lease the software on-demand. The domain of high assurance distributed systems has focused greatly on the areas of fault tolerance and dependability. In a multi-tenant context, it is particularly important to store, manage and provision data services to customers in a highly efficient and dependable manner due to a large number of file operations involved in running such services. It is also desirable to allow a user group to share and cooperate (e.g., co-edit) on some specific data. In this paper we present a dependable data provisioning service in a multi-tenant Cloud environment. We describe a metadata management approach and leverage multiple replicated metadata caching to shorten the file access time, with the improved efficiency of data sharing. In order to reduce frequent data transmission and data access latency, we introduce a distributed cooperative disk cache mechanism that supports effective cache placement and pull-push cache synchronization. In addition, we use efficient component failover to enhance the service dependability whilst avoiding negative impact from system failures. Our experimental results show that our system can significantly reduce both unused data transmission and response latency. Specifically, over 50% network transmission and operational latency can be saved for random reads while 28.24% network traffic and 25% response latency can be reduced for random write operations. We believe that these findings are demonstrating positive results along the right direction of resolving storage-related challenges in a multi-tenant Cloud environment.**

***Keywords*-Multi-tenant Cloud; Cloud Storage; Metadata; Co-operative Disk Cache**

I. Introduction

The increasing maturity of Internet and virtualization techniques is leading to a new delivery and business model – Software as a Service (SaaS) in multi-tenant Cloud [10] [15]. Specifically, customers only need to lease the software or service on-demand rather than purchasing and maintaining the software suite, because all service deployments and managements are undertaken by software vendors. In recent years, it is experiencing rapid growth of applications from both industry and academia, such as Google Apps [6], Citrix XenApp [2], CloudAP [28] etc. Following *pay as you go* philosophy, users will be charged according to the accumulative time or resource consumption. Apart from effective software execution mechanism, how to efficiently store, manage and provision dependable data and file access service is of significant importance due to a large number of file operations during the service runing.

Network file system (NFS) [20] is adopted by many systems as their data storage. Despite the rapid and efficient file operations provided by NFS, the upgrading cost and scalability issues are still big concerns. In particular, the increasing expense to purchase hardware servers will become a heavy burden to service providers especially when current cluster capacity cannot satisfy the bursting data storage requirements. System scalability to handle concurrent requests and performance issues [29] [26] is also a non-negligible factor to be considered. Additionally, new personal storage model is advocated, in which it jointly leverages different user devices such as PC, laptop and smart phones etc. [12]. For instance, Eyo [23] proposes a mechanism of such storage sharing pattern and it could make the most use of user's storage capacity. However, unstable network condition in 3G and WiFi often leads to degraded and unaccepted *degree of dependability*. In fact, a reliable and effective service could reduce the economic impact and service degradation for providers and consumers respectively. Furthermore, Cloud storage has obviously become a practical facility [22] [27] because it elastically provisions a large number of storage capacities through APIs or SDKs, making customers free from dealing with scalability and failure issues by themselves. However, data privacy becomes a big issue for Cloud Storage. Users might be reluctant to put their private or confidential data into public storage spaces. Moreover, it is also highly desirable in multi-tenant environment to allow a *user group* to share and cooperate (e.g., co-edit) on some specific files. However, limitations of the approaches mentioned above would impede the fully-utilized file sharing and collaborative operations.

To this end, combining Cloud storage with user individual devices whilst providing effective data sharing mechanism becomes an appropriate candidate solution. A user could determine his own data distribution – private data stored on local devices and the remaining data in the Cloud. In this context, some challenges are still far from settled: Firstly, typical key-value pair storage in Cloud storage or distributed file system lacks of mutual relationship information among different pairs. Highly required file attribute descriptions are not included either. Secondly, reducing the resource consumption (especially for network traffic) is a very critical goal. This is due to the fact that network bandwidth is a scarce resource in multi-tenant business model. The reduction of network traffic indicates the

1530-2059/16 $31.00 © 2016 IEEE
DOI 10.1109/HASE.2016.26

cutting down of economic expense with improved QoS and increased system capacity. Thirdly, operational latency has become one of the principal aspects of dependable service provisioning, especially for interactive applications. In general, data transmission delay is currently the norm rather than the exception because the communications among storage devices are mainly domain-crossing. Furthermore, data synchronization and data consistency are also extremely indispensable during file co-operations in which multiple tenants or software processes might simultaneously modify a specific data.

In order to deal with these problems, we design and implement D^2PS – a dependable data provisioning service. The core philosophy is to provision an effective data access and efficient data sharing mechanism among collaborative users or softwares. The data provenance derives from a unified data view based on both Cloud storage and individual devices. Firstly, we adopt a novel metadata to combine both file attributes and the tree-based hierarchy of all user directories and files no matter which the original structure is (key-value or tree). Additionally, metadata local caching and consistency guarantee are advocated to facilitate the software process initialization and rapid data access. Furthermore, we use distributed cooperative disk cache to buffer the frequently-used data. All above caching mechanisms follow a two-tier caching architecture which consists of a centralized coordinator and multiple proxy daemons on every execution nodes. The effective caching reduces the communication frequency and unused data transmission. In particular, we leverage hybrid pull-push approach to achieve bidirectional cache data synchronization, and incremental transaction with queue-based flow control to resolve conflicts incurred by simultaneous modifications. Specifically, light-weight checkpointing is also used for rapid component failover, providing effective and dependable execution environment. Our system is based on iVIC [25], Alibaba ECS [3] and OSS [7], and the experimental results show that our system significantly reduce both network traffic and the response latency. Specifically, over 50% network data transmission and operational latency can be diminished for the random read while 28.24% network traffic and 25% response latency can be reduced for random write operations. The major contributions in this paper can be summarized as follows:

- An effective metadata management mechanism including a coordinator-follower based data consistency approach, and conflict resolving with incremental transaction and queue-based flow control.
- A two-tier caching approach based on cooperative disk cache with a pull/push synchronization mechanism.
- A dependable data provisioning service with transparent component failover, providing a unified data view combining both Cloud storage and individual devices.

The remaining sections are structured as follows: Section II presents the problem and system overview; Section III describes the metadata management approach and Section IV depicts the detailed design of cooperative disk cache; The evaluation is presented in Section V; Section VI shows related work and we finally conclude the paper with future work.

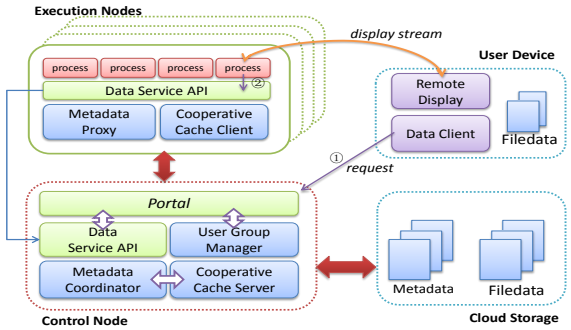


Fig. 1: System architecture overview

II. Problem Definition and System Overview

In multi-tenant SaaS execution environment, key objectives in dependable data provisioning are: data sharing within a group, data isolation among different groups, data or file operation performance guarantees with low latency network transmission amount, and reliable service running with minimized impacts on customers SLA etc.

Firstly, in order to provision each SaaS tenant a dependable and transparent data view of the global storage resources whilst protecting the user privacy, we adopt the notion – *user group* – a dependable and securable means of admission control. Data inside a user group could be shared and co-operated among different users, while users in other groups are disallowed to get access to these files and data. All these mentioned above are handled by the User Group Manager shown in Figure 1. Requests are sent to the control node and each request is marked with a group label. The centralized controller will differentiate them and enable an isolation of files for different groups. For convenience, we use *user* in the remaining sections to comprehensively represent *user group*.

Figure 1 depicts the system overall architecture and the core design idea. The portal provides a web-based user interface(UI) for administrating, files and data visualization, event and status monitoring etc. The built-in scheduler in the portal is actually the delegator, responsible for message forwarding, routing and communications among SaaS execution nodes and storage sources (Cloud and user's personal device). The data provisioning service APIs will be invoked by file operation requests. In fact, two vital underlying processes – metadata coordinator and data cache server could offer direct and complete information to support file loading and operations. Data operations mainly occur in two different ways: through user direct access and by running software processes (step 1 and 2). On each execution node, two daemon processes – metadata proxy and cooperative cache client are launched. Metadata proxy will directly handle metadata-related requests based on its locally-cached replica. In this manner, the load and pressure of metadata request handling on the centralized controller node will be dispersed onto each metadata proxy, resulting in the mitigation of single point failure. The relevant techniques will

be discussed in Section III. After full data indexing from metadata has been constructed, cooperative cache client process will deal with and forward the data operation requests to the data cache server. The holistic resources among all execution nodes will be utilized to constitute a distributed cache resource pool and each cache client works cooperatively as dual roles – both cache producer and customer. The specified system design will be depicted in Section IV.

III. Metadata Management

The data-structure in Cloud is typically stored in the form of key-value pairs, lacking of mutual structural relationship among different pairs and highly-required file attributes descriptions. These characters are quite different from tree-based structure in Linux file system. To satisfy the requirements, we propose a novel metadata to record the user directory tree and file attributes. The metadata and the corresponding files are stored separately in Cloud storage for dependability considerations. Furthermore, metadata cache is conducted in each node to accelerate the data loading and accessing. However, inconsistency and conflict might appear when multiple replicas are concurrently modified. Therefore, the metadata consensus and conflict resolving are significantly important.

A. Metadata Structure

As mentioned in Section II, each user group will have an exclusive metadata. The logical storage structure is a tree with multiple branches and it is persistently stored as an object in the Cloud while file data pertaining to the user will stored separately for resiliency. In one specified tree, each node represents a directory entry or a piece of file data. The directory entry is a recursive notation which contains the parents and all child directories. Basically, file attributes are recorded including the last modification time, access time, version, location and replica information etc. Table I depicts the attribute list of a file entry in detail.

TABLE I: The structure of the pseudo metadata

attribute name	attribute meanning
st_mode	file type and permissions
st_inode	inode number
st_nlink	number of links
st_uid	user ID of user
st_gid	group ID of user
st_size	size in bytes
st_atime	time of last access
st_mtime	time of last modification
st_ctime	time of file creation

B. Consistency Maintenance

We adopt a *coordinator-follower* architecture to guarantee the metadata consistency among multiple nodes. Firstly, the coordinator is a core component that is responsible for timely fetching and synchronizing the metadata once the original data is changed. It is fulfilled by periodically sending *FullMetadataInfoRequest* or event-driven notification from the data source. Accordingly, follower is a daemon process, located

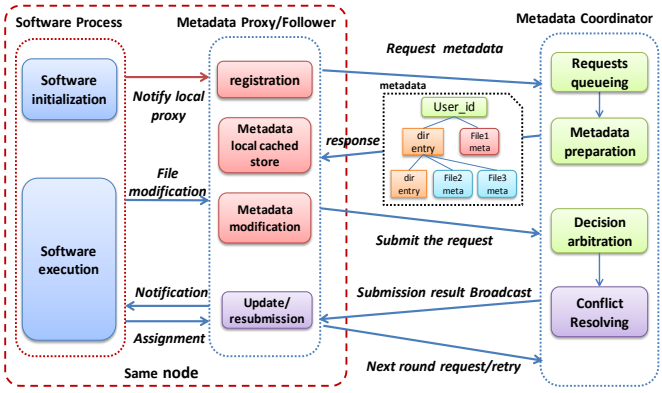


Fig. 2: The basic workflow of coordinator-follower architecture: consistency maintenance and conflicts resolving.

in each execution node where the user software processes run. The daemon firstly petitions the coordinator for full metadata if it does not have a replica in its initialization and then repeatedly retrieves the updated data segments. It is noteworthy that only the coordinator itself is the mutator, which can conduct changes to the metadata permanent store. The coordinator also has to aggregate full information once the source file updates and it will propagate these updates to all replicas hold by followers. Meanwhile, some modifications might happen during software running period and they are mainly composed of new node adding, removing, and properties changes of existing nodes. Because follower operations are conducted based on its local cached metadata replica, conflicts will be emerging consequently. Figure 2 illustrates the workflow of our proposed model. Once the software starts running, the daemon at the software execution node sends metadata request to the coordinator. On receiving the demand, the coordinator records it within a request queue. Meanwhile, it locally searches the relevant metadata and if the metadata is not found, a synchronization is conducted (by *FullMetadataInfoRequest*) to fetch the required data from Cloud storage or personal terminals. The initiation phase finishes after the latest metadata is re-synchronized in both coordinator and related followers.

C. Conflict Resolving

Conflicts are resolved optimistically by the coordinator in our system. In particular, the *incremental transaction* [21] is used in which we accept all but the conflicting modifications independently. The unaccepted changes parts will be returned to the metadata follower and some measures should be taken in the next operation round with modifications re-submitted. Besides, metadata follower could also choose *gang transaction* mode, for the purpose of completely atomic transaction in order to accept all changes. However, it is very likely to incur starvation and even deadlock if no more requests could be satisfied. To prevent this but still keep parallelism, we set a configurable concurrency threshold and use a *waiting queue* inside the coordinator to enqueue the incoming requests

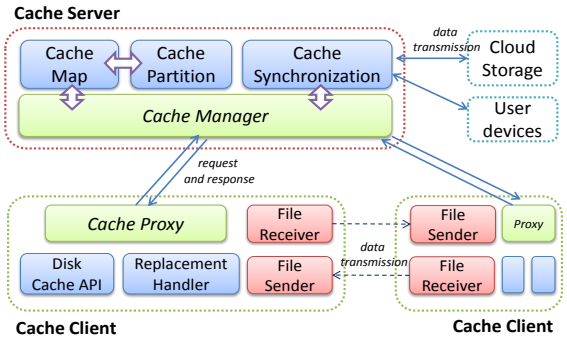


Fig. 3: The design of cooperative disk cache

when the request number surpasses the threshold. As shown in Figure 2, concurrent modifications will be handled during the decision arbitration through the proposed transaction with queue-based flow-control. Eventually, the coordinator forwardly broadcasts the changed metadata to all relevant followers. In this manner, those potential conflicts might be detected as early as possible.

D. Dependability with Failover

Although the two-tier metadata coordinator and follower architecture could naturally balance request loads into separate proxies on different execution nodes, the frequency of metadata requests will still grow dramatically with the soaring user number. Therefore, the single metadata coordinator process will become the potential single-point bottleneck and very likely to be susceptible to failures. For this reason, we actually replicate the single process three times. Specifically, one replica is active and the others are warm-standby. The leader election and the high available mechanism are implemented by using Raft [18]. Furthermore, considering the acceptable metadata size and restore costs, we merely conduct a light-weighting checkpointing for each meatadata periodically when repeatedly updating with the Cloud storage. Whenever the active fails, a standby replica will take over and start to failover according to the checkpoint.

IV. Cooperative Disk Cache

Apparently, nodes are directly connected within the same rack or the same computing cluster. This leads to the fact that data transmission rate among those servers is much faster than that among Cloud storage or personal device storage through WAN. This phenomenon inspires us to turn to cache mechanism for much balanced file access. However, traditional cooperative memory caching [11] [14] is too small to hold the data. Hence, we advocate a *distributed cooperative disk caching* approach to satisfy large-scale user data caching demands for applications. Specifically, all nodes within the cluster could be fully utilized as a whole caching pool.

A. Basic Idea

As shown in Figure 3, cooperative disk cache follows a server/client architecture. The cache server daemon is located

Algorithm 1 Placement and Replacement Algorithm

Input:
$Cache_{total}$ ← the size of the disk cache
$Cache_{occupied}$ ← the available size of the disk cache
$size$ ← the size of the data to be cached
$CacheList$ ← a list of the cached data according to *cacheMap*

1: **if** $size + Cache_{occupied} <= Cache_{total}$ **then**
2: find a *diskCache* in a candidate node by using a plugin cache placement algorithm
3: *diskCache* ← file data
4: **else**
5: **for** each *data* in *CacheList* **do**
6: *MetadataList* ← look up the metadata of *data* in *MetadataProxy*
7: **end for**
8: *targetedCache* ← find the $data_{cached}$ with the least access times in *MetadataList*
9: *targetedCache* ← replacing with file data
10: **end if**
11: update *cacheMap*

in a particular node and acts as a dominant controller of all distributed cache blocks. It handles cache requests, decides the cache placement and indexing, and manages the whole cache life-cycle. Specifically, Cache Manager handles all requests from cache clients. CacheMap maintains the mapper relationship containing cache partition, partition size, and the partition position distributed among the cluster. we adopt a load balance partition algorithm to instruct cache placement (Section IV-B), and the results will be recorded in the mapper. On the other hand, the cache client process resides on each node and cooperates with peer clients. In fact, the client provides two-folded *proxy* functionalities – receiver and sender. It not only manages the specific caches stored on its pertaining node, but communicates with other cooperated peer clients as well.

Consider a software running in the SaaS system which plans to modify a file data. It initially notifies the cache client through an API and the client will then delegate the caching request like a *proxy*. It firstly checks the local cache and determines whether the cache is hit or not. If not, the client will route the request to the cache server. Thereafter, the cache server looks up to the map in order to find the target cache position. However, if the required file cache is not in the distributed cache pool at present, the server will mandatorily ask for pulling the source data by invoking *FullDataRequest* from Cloud storage or user devices immediately. Subsequently, the cache placement will be conducted with cache mapper information updated. After getting the response from cache server, the cache client will communicate with the peer client on the target node to obtain the required cache data.

B. Cache Placement and Replacement

As mentioned above, deciding which node the incoming fetched data should be cached is essentially important. The cache partition will be refreshed after each placement round and recorded in the cache mapper. In general, the cache client delegates and routes the request (that new data needs to be cached) to the cache server, trying to get an available disk space. The server will make assignment decisions based on criteria such as data locality, load balance according to the current caching information. Moreover, due to the limited capacity of system disk, only a proportion of disk resource can be leveraged as memory auxiliary cache according to a pre-defined threshold. The value is set per node and will be also considered in the cache placement. In order to maximize the cache utilization, cache replacement is designed to evict some stale data and make sufficient rooms for the new one.

As demonstrated in Algorithm 1, if the new data size is within the available space, the data could be filled directly into the global cache pool and is very preferable to local disk cache for rapid access consideration (line 1-2). If there are insufficient rooms to hold the incoming request, cache client will traverse the metadata to find a cached data with the least recent used (LRU) frequency. Eventually, the required data will take it over (line 3-8).

C. Cache Synchronization

To ensure the system availability, we perform data consistency and bidirectional synchronization between cooperative disk cache and data source. The core idea of push-pull method is illustrated in Algorithm 2. Firstly, the modifications will be push back to its source once a file is changed. For example, if a file data is edited, it subsequently results in some re-writes to the cooperative disk cache. In some cases, the metadata proxy will also submit a request, trying to update the data attributes in metadata (as mentioned in Section III-B). Finally, the changes will be written back to the data source. During the whole process, the synchronization is conducted by passive push to the source storage. Secondly, if files are updated from personal devices or Cloud storage, the metadata coordinator will fetch a complete information of metadata periodically through *FullMetadataInfoRequest* or after being notified by events from storage sources (as mentioned in Section III-B). Afterwards, a comparison will be immediately conducted to calculate the differences between the incoming metadata and the previously cached one. If a difference exists, the cache server will send *DataRequest* to the remote data source for new data and use it to replace the stale caches.

D. Dependability with Component Failover

The cache server and cache clients play important roles in the cooperative disk cache management architecture. Cache server is particularly vital because it not only maintains the global information about cache mapping but also undertakes core functionalities such as cache placement and communication. Therefore, once failure happens, these daemon processes must appear to rapidly recovery, without noticeable changes to the provisioned service. To achieve this, we have designed these daemons to be *soft state*, indicating that a failed component could completely recover from information held by other relevant components without heavy checkpointing and rollback overheads. Specifically, the states for the cache server mainly include full cache mapper information and active clients list. These could be collected and finally refilled from each connecting client after cache server process restarts. Afterwards, the new cache server will send *FullDataRequest* and new threads are launched in order to rapidly re-fetch the corresponding source file data. Due to the *stateless* character, the cache client process could directly reboot and complete the failover after reconnecting to the cache server.

Algorithm 2 Push-pull Bidirectional Consistency Algorithm

Definition:
$MetadataList_{cached}$: currently cached metadata
Γ: a fixed time-interval
$MetadataList_{new}(\Gamma)$: a full metadata at time interval Γ
D(Γ): requested file data from sources at time interval Γ

//push - modification happens
1: **if** *diskCache.modified* is ***True*** **then**
2: update $MetadataList_{cached}$
3: write back to the source storage
4: **end if**

//pull - each time when fetching new metadata
1: **for** each *Metadata* in $MetadataList_{new}(\Gamma)$ **do**
2: **if** *Metadata* not in $MetadataList_{cached}$ **then**
3: $MetadataList_{cached} \leftarrow$ *Metadata*
4: *diskCache* $\leftarrow$ D(Γ)
5: update *CacheMap*
6: **else if** *Metadata.modificationTime* is newer **then**
7: update *diskCache* with D(Γ)
8: update $MetadataList_{cached}$
9: **end if**
10: **end for**

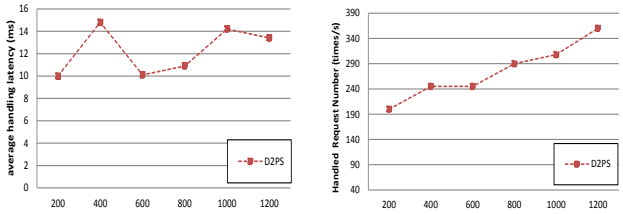
V. Experiments and Evaluation

A. Experimental Setup

The evaluation environment consists of three parts: application platform(deployed in both private Cloud and public Cloud), Cloud storage and users personal devices. Specifically, the private cloud is constructed based on iVIC [25], consisting of 32 physical servers connected within local area network. Meanwhile, we deploy our system in another 32 virtual machines using the Alibaba Cloud ECS [3] as the public Cloud environment. Overall 200GB Cloud storage spaces of OSS [7] are used, providing GET/PUT operations interfaces. The specified configurations are listed in Table II where one server is used as the control node and other servers act as execution nodes. In our evaluation, we prepare 16 types of random generated files with different sizes which range from 8KB to

TABLE II: Experimental Setup

Types	Configurations	OS version	Bandwidth
Server	Intel Xen E5640 2.4GHz CPU 2.5GB DDR2 ECC memory, 250GB SCSI hard disk,	debian 6.0 kernel 2.6.31	1Gbps
ECS	2 cores, 4GB DDR2 memory, 40GB disk,	Ubuntu 14.04	1Gbps
User Device	Intel Core(TM)-860 2.8GHz CPU 4.0GB DDR2 ECC memory 500GB SCSI hard disk	window 7	100Mbps

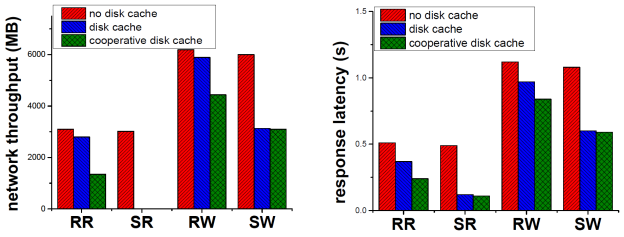


(a) Handled Request Number (b) Request Handling Latency

Fig. 4: The system performance for request handling

TABLE III: The cache hit number of 2000x random read

	local disk cache hit	cooperative disk cache hit	total request number
no disk cache	0	0	2000
single disk cache	289	0	2000
cooperative disk cache	289	715	2000



(a) The network traffic (b) The response latency

Fig. 5: The performance improvement with cooperative disk cache under 2000x random read(RR), subsequential read(SR), random write(RW) and subsequential write(SW) workload

5MB, following nth-power of two approximately. These files are stored in OSS and user devices respectively.

In order to provision the data service APIs to running softwares and users, we implement the system by extending the userspace file system [4] basic APIs while combining the two-tier caching mechanism of both metadata and cooperative disk cache. We implement the following APIs including: *getattr*, *rename*, *mkdir*, *read and write*, *rm and rmdir* etc.

The following metrics are considered: **1) request handling rate and latency:** the handling effect under a fix submission rate; **2) cache hit ratio:** the proportion of targeted cache trials; **3) network traffic:** the total amount of data transmission through network; **4) response latency:** the time duration from the time when requests are sent out through our data provisioning APIs until the file operation finishes. In order to manifest the caching effects, we mainly compare three different deployment approaches: no cache, disk cache, and proposed cooperative disk cache separately, while the simulated data API calls could be categorized into four types – 2,000 x random read or random write, 2000 x sequential read or sequential write. To emulate the extreme user scenario, these file operations are conducted concurrently by multiple threads API calls. We set the concurrent number to be 50 and calculate the average for the results.

B. Request Handling Effect

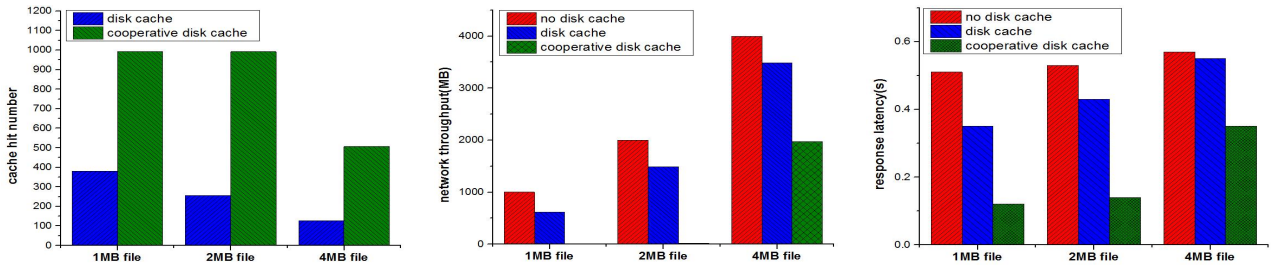
In this experiment, we measure the concurrent request handling effect under different concurrent submission rates (requests per second). It is observable from Figure 4 that the handled number can grow with the increasing incoming requests although not all requests could be handle at a time. This is because our proxy-based mechanism could effectively handle requests on each execution node in parallel and the queue could mitigate the surging requests in a flow-control way. The similar phenomenon of handling latency can be found due to the same reason. The results illustrate that an enhanced scalability of request handling could be achieved based on our system architecture.

C. Operational Performance Comparison

*1) **Cache Hit Ratio**:* In random read experiment, we conduct 2,000 times read operations. It is observable from Table III that the cache hit ratio is only 14.45% when adopting disk cache. Another 715 more cache hit events could be achieved by leveraging cooperative disk cache. As a result, roughly 50% cache hit will no doubt promote the data loading rate, with user experience greatly improved.

*2) **Read Performance**:* Figure 5(a) depicts the network traffic during file operations. For random read, the total network traffic is 3095MB due to 2000 times file fetch at each time with 1.54MB median file data size. In comparison, disk cache saves approximately 300MB when reading the same amount of files. Additionally, the traffic reduction of network could even reach more than 50% with cooperative disk cache compared with disk cache mechanism. Similarly, the average random data read latency is shown in Figure 5(b). It takes 0.51s without any data cache mechanism and the latency decreases to 0.37s and 0.24s respectively with disk cache and cooperative disk cache deployed. The reason for this is that some data could be prefetched and cached in the local disk and we do not have to load those data from remote storage every time. Moreover, because the cooperative disk cache leverages servers in the cluster as a whole distributed disk cache pool, abundant disk resources could be definitely extended to support more loaded data, thereby significantly increasing the cache hit whilst reducing the operational latency.

In terms of the sequential file read of the same size file, the network transmission amount is 3019MB, with a slight decrease compared to the random read experiment. In fact, random read will consume a little bit more resources when



(a) The number of cache hit (b) The network traffic (c) The average response latency

Fig. 6: The performance impact of 1MB, 2MB and 4MB source file data under 1000x random read

fetching the original data. We could also observe from Figure 5(a) that the network traffic is reduced to only 1.5MB by leveraging disk cache or cooperative cache and the latency also decreases from 0.49s to 0.12s and 0.11s respectively (see Figure 5(b)). The significant improvement is due to the fact that the file data only needs to be cached at first and the subsequent operations could directly get access to the data from the local cache.

*3) **Write Performance**:* As for the random write performance, 6190MB network flows are consumed without any cache. The amount is reduced by 4.77% using disk cache and 28.24% network traffic could be saved with the cooperative disk cache. Correspondingly, the response latency drops from 1.12s with no cache to 0.97s (13.4% reduction) and to 0.84s (25% reduction) respectively under the other two conditions. Meanwhile, the same phenomenon could also be observed for subsequential write. The absolute decreases under these two random write scenarios are approximately same as the reduction when randomly reading. This reduced gap is attributed to the read effect among different approaches. In fact, each write operation round is comprised of three different phases: file open and read, write and save operation, and write-back. No matter which type of cache is adopted, there is little difference in the second phase because all operations are based on one local specific file. In the third phase, for the sake of consistency, each file has to be written back to the source leading to the similarity among different approaches.

D. Impact on Cooperative Cache Performance

To comprehensively explain the cooperative cache mechanism, another three experiments are further conducted. We prepare files on OSS and user local storage with 1MB, 2MB and 4MB size in different groups. In addition, the upper threshold of disk cache in each server is configured by 4MB and 1,000 times random read operations are carried out.

Figure 6(a) demonstrates the cache hit number. The number will decrease according to the increased file size because the smaller the file size is, the more probability of loading files into the cache will be achieved. For example, the hit number will be doubled if the file size shrinks half from 4MB. Furthermore, the cache hit ratio will stay 99.2% if the file is less than 2MB, indicating that the distributed cooperative disk cache pool will facilitate the cache effect. Even the file size approaches the cache upper bound, the hit ratio (50.7%) is still much larger than the ratio in original disk cache cases. Moreover, the cache hit effect will have a direct impact on the file operational performance. Obviously, the proposed mechanism outperforms the others as shown in Figure 6(b) and Figure 6(c). The reduction of the network traffic could reach 50% at most by using distributed cache mechanism. Meanwhile, the corresponding response latency also significantly decreases by 76.4%, 71.6% and 36.7% respectively under different experimental configurations.

E. Failure Recovery Effect

The recovery from master failures is evaluated by fault injections. We re-conduct the experiment mentioned in Section V-C2 and randomly kill the active meta coordinator and cache server process every 30 seconds with the mean time to recovery (MTTR) measured. The service could be recovered within 5 seconds on average with a standard deviation of 0.4 second. Additionally, the average response latency of random write increase from 0.84s to 1.02s while the average latency increase from 0.59s to 0.76s under subsequential write. The slight delays are mainly due to the caching service unavailability during the frequent failovers. Nevertheless, these catastrophes are very atypical as the adopted fault injection scenario is so harsh that few probabilities exist in real-life systems.

VI. Related Work

DepSky [9] and BlueSky [24] are network file system which store their data persistently in a Cloud storage vendors (Amazon S3 [1], Windows Azure [8] etc.) and allow users to take the advantage of the reliability and large storage capacity from Cloud providers. However, both DepSky and BlueSky mainly focus on invariants like availability. How to share storages among personal devices is actually not mentioned. In multi-tenant SaaS Cloud, invariants such as reduction of network traffic and response latency are significantly important to user experiences, and thus have to be considered. In terms of multiple device storage sharing, AFS [13] pioneers the use of a single namespace to manage a set of servers. It requires that client to be connected with AFS servers and clients cache files that have been hoarded. BlueFS [17] and Ensem-Blue [19] handle a variety of modern devices and use a peer-to-peer update dissemination to improve the performance. ZZFS

[16] focuses mainly on the low-power connection with an add-on hardware. However, they only handle the storage sharing issues in LAN while ZZFS relies on extra hardware.

Furthermore, Google Docs [6] is a distinguished web-based SaaS product which can provide users with not only editing service but document sharing service among multiple users as well. All documents are stored in Google Drive [5]. However, it is an internal data storage service and can not be used to build experimental SaaS storage. Saga [22] is a userspace file system based on Amazon S3 [1] and it adopts fixed size block as storage unit to reduce the storage occupation. DO-LRU cache replacement strategy is implemented in order to improve the request performance whilst reducing the costs of using Cloud storage. Despite this, Saga only considers single client design, ignoring the common scenarios of multiple personal device management. Besides, cooperative cache is more effective than the ordinary cache mechanism, and it can be derived from [11] and [14]. They put part of the memories of each workstation in the cluster together to form a larger global collaboration cache. In this way, the cache system could increase the cache hit ratio while reducing the number of disk access. However, the cooperative cache is mainly established based on memory. In SaaS platform, due to the large number of users, memory cache replacement will be extremely frequent, resulting in reduced benefits from previous approaches.

VII. Conclusions and Future Work

This paper presents a dependable data provisioning service in the multi-tenant Cloud environment. It is highly desirable to allow a user group to share and cooperate (e.g., co-edit) on some specific data or files. Therefore, we describe an effective metadata management approach in which we take both data relational structure and data attributes into account, and leverage multiple replicated metadata caching to improve the efficiency of data sharing and data access. Furthermore, we advocate a distributed cooperative disk cache mechanism to decrease the data transmission and the file access latency among different storage provenances. The scalability and parallelism issues such as effective concurrency handling, data consistency, conflict resolving, efficient component failover are also addressed in this paper. The experimental results show that our system can significantly reduce both the network traffic and the response latency. Specifically, over 50% network traffic and operational latency are reduced in the random read experiment while 28.24% network traffic and 25% response latency are reduced for random write operations. In the future, we will further improve the current cache mechanism considering pre-fetching file blocks based on user behavior predictions. The historical information will increase the cache hit ratio and reduce the response latency. We will also evaluate the failover effects under more sophisticated scenarios in which multiple component failure combinations might occur simultaneously.

Acknowledgements

This work is supported in part by China 973 Program (No.2011CB302602), China 863 program (2015AA01A202), HGJ Program (2013ZX01039002-001), Fundamental Research Funds for the Central Universities and Beijing Higher Education Young Elite Teacher Project (YETP1092), and NSFC(91118008, 61170294),. Tianyu Wo is the corresponding author.

References

[1] Amazon S3. https://aws.amazon.com/s3/.
[2] Citrix XenApp. http://www.citrix.com/xenapp.
[3] ECS. http://www.aliyun.com/product/oss/?lang=en.
[4] Fuse. http://fuse.sourceforge.net/.
[5] Google app engine. http://code.google.com/appengine.
[6] GoogleApps. http://www.google.com/apps.
[7] OSS. http://www.aliyun.com/product/ecs/?lang=en.
[8] Windows Azure. https://azure.microsoft.com/.
[9] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS) 2013*, 9(4):12, 2013.
[10] P. Buxmann, T. Hess, and S. Lehmann. Software as a service. *Wirtschaftsinformatik*, 50(6):500–503, 2008.
[11] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of USENIX OSDI 1994*, 1994.
[12] S. Dhumbumroong and K. Piromsopa. Personal cloud filesystem: A distributed unification filesystem for personal computer and portable device. In *Proceedings of JCSSE 2011*, pages 58–62. IEEE, 2011.
[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.
[14] A. Leff, P. S. Yu, and J. L. Wolf. Policies for efficient memory utilization in a remote caching architecture. In *Proceedings of ICPDIS 1991*. IEEE.
[15] D. Ma. The business model of" software-as-a-service". In *Proceedings of IEEE SCC 2007*, pages 701–702. IEEE, 2007.
[16] M. L. Mazurek, E. Thereska, D. Gunawardena, R. H. Harper, and J. Scott. Zzfs: a hybrid device and cloud file system for spontaneous users. In *Proceedings of USENIX FAST 2012*, 2012.
[17] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of USENIX OSDI 2004*.
[18] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX ATC 2014*, page 305, 2014.
[19] D. Peek and J. Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *Proceedings of USENIX OSDI 2006*.
[20] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.
[21] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of ACM EuroSys 2013*, 2013.
[22] W. Shi, D. Ju, and D. Wang. Saga: A cost efficient file system based on cloud storage service. In *Economics of Grids, Clouds, Systems, and Services*, pages 173–184. Springer, 2012.
[23] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and M. F. Kaashoek. Eyo: Device-transparent personal storage. In *Proceedings of USENIX ATC 2011*, 2011.
[24] M. Vrable, S. Savage, and G. M. Voelker. Bluesky: a cloud-backed file system for the enterprise. In *Proceedings of USENIX FAST 2012*.
[25] T. Wo, C. Hu, J. Li, and J. Huai. Netros: A virtual computing environment towards instant service of network software. In *Proceedings of IEEE SKG 2012*, pages 24–31. IEEE, 2012.
[26] R. Yang, I. S. Moreno, J. Xu, and T. Wo. An analysis of performance interference effects on energy-efficiency of virtualized cloud environments. In *Proceedings of IEEE CloudCom 2013*. IEEE, 2013.
[27] W. Zeng, Y. Zhao, K. Ou, and W. Song. Research on cloud storage architecture and key technologies. In *Proceedings of ICIS 2009*, pages 1044–1048. ACM, 2009.
[28] Y. Zhang, R. Yang, T. Wo, C. Hu, J. Kang, and L. Cui. Cloudap: Improving the qos of mobile applications with efficient vm migration. In *Proceedings of IEEE HPCC 2013*, 2013.
[29] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proceedings of the VLDB Endowment*, 2014.