MetaStorage: A Federated Cloud Storage System to Manage Consistency-Latency Tradeoffs

David Bermbach, Markus Klems and Stefan Tai
Institute of Applied Informatics and
Formal Description Methods (AIFB)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Email: firstname.lastname@kit.edu

Michael Menzel
Forschungszentrum Informatik (FZI)
Karlsruhe, Germany
Email: menzel@fzi.de

***Abstract*—Cost and scalability benefits of Cloud storage services are apparent. However, selecting a single storage service provider limits availability and scalability to the selected provider and may further cause a vendor lock-in effect. In this paper, we present MetaStorage, a federated Cloud storage system that can integrate diverse Cloud storage providers. MetaStorage is a highly available and scalable distributed hashtable that replicates data on top of diverse storage services. MetaStorage reuses mechanisms from Amazon's Dynamo for cross-provider replication and hence introduces a novel approach to manage consistency-latency tradeoffs by extending the traditional quorum (N, R, W) configurations to an (N_P, R, W) scheme that includes different providers as an additional dimension. With MetaStorage, new means to control consistency-latency tradeoffs are introduced.**

I. Introduction

Over the last few years the Cloud Computing market has grown tremendously and frequent new service offerings are emerging steadily. In particular, there is a large number of Cloud storage services, each focusing on different capabilities and guarantees. To satisfy availability and scalability needs of most Cloud-based applications, NoSQL databases have become very popular, owning the highest share of Cloud storage offerings [1], [2]. Besides, in-memory databases or distributed relational database clusters are common alternatives, providing high-performance and consistency guarantees respectively. The choices are many, but vendor lock-in is still an issue as Cloud storage offerings tie customers to one particular offering due to immense switching costs for data migration. Even when switching between similar NoSQL stores the effort can be extensive due to large databases and varying service implementations. Moreover, relying on a single Cloud storage provider results in an avoidable single point of failure and challenges high-availability and reliability aspirations [3]–[7]. Despite geographically distributed data centers and advanced, robust Cloud storage technology a provider can turn into the single point of failure and cause a business risk, e.g. due to downtimes or even going out of business. Depending on the technologies, business know-how as well as the decisions of a single provider increases the vulnerability of every customer and puts the reliability of a system at risk. Especially when running business critical systems in the Cloud, business success and competitive advantage heavily rely on the activities of a single provider. The only way to overcome the dependency on a single provider is to replicate data to multiple providers in order to maintain the possibility for immediate provider switches and to mitigate downtimes.

The remainder of the paper is structured as follows: First, we introduce MetaStorage, a Cloud storage federation system, and describe its design and implementation details as well as the additional parameters we introduced to balance consistency-latency tradeoffs. Afterwards, we present the results of a system evaluation regarding consistency, availability and latency. Finally, we discuss the system's weaknesses and strengths and end with a conclusion.

II. MetaStorage

MetaStorage is a highly scalable, highly available, distributed hashtable, layered on top of different Cloud storage providers. For this purpose, MetaStorage reuses mechanisms from Amazon's Dynamo [8], but elevates these for cross-provider data replication to maximize scalability, availability, vendor independence and fault tolerance. As MetaStorage leverages Cloud storage services which typically offer only eventual consistency it is obvious that it can never fully guarantee strict consistency to the outside world. So, by reusing Dynamo's design principles MetaStorage introduces an eventually consistent scheme itself.

MetaStorage replicates data across several providers instead of using machines of a single provider only. By integrating diverse Cloud storage providers, MetaStorage extends traditional quorum systems that use (N, R, W) configurations to balance not only consistency-availability but also to balance consistency-latency tradeoffs. Now, we can still use (N, R, W) but also add the dimension of providers as an additional knob to tweak consistency-latency tradeoffs. We suggest novel (N_P, R, W) configurations where $N_P \geq 1$ is the total number of replica $|\mathcal{N}_P|$ that are hosted with the set of n providers. The providers host a set of replica, formally defined as $\mathcal{N}_P = \mathcal{N}_1 \cup ... \cup \mathcal{N}_n$ where each provider $i \in \{1, ..., n\}$ hosts $|\mathcal{N}_i|$ replica.

In the following we will present the MetaStorage system, its design and implementation.

A. MetaStorage Architecture

The MetaStorage architecture (figure 1) is based on nodes which act as wrappers for Cloud storage services. A set of nodes is aggregated within a Distributor which includes all functionality to replicate and retrieve data as well as assert availability of replica. To avoid the Distributor becoming a bottleneck we attached a Coordinator component to each Distributor which is responsible for periodically exchanging state between Distributors. MetaStorage components internally communicate using an asynchronous messaging protocol which can be seen as a subset of the staged event-driven architecture (SEDA) [9]. The main advantage of SEDA is that it degrades gracefully under heavy load as the overhead for thread synchronization stays constant no matter how many requests have to be processed per second. This is the reason why it was also internally used within Dynamo and reused in our context.

In the next section we will give an in-depth description of all components within the MetaStorage architecture.

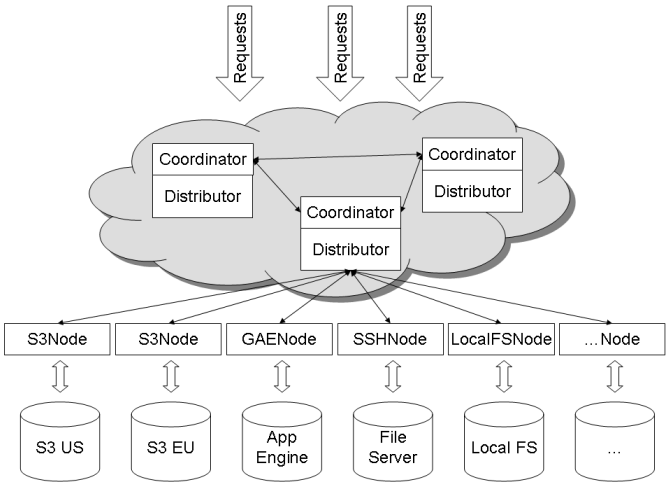

Fig. 1. High-level overview of the MetaStorage architecture

B. MetaStorage Nodes

Within the MetaStorage system the nodes are situated at the lowest layer. Their most important task is to offer a generic interface to the Distributor so that all technical details of the underlying infrastructure are hidden. Thus, they are basically wrappers for Cloud storage services like Amazon S3 (see also [10] and [11]). So far, we have built nodes for Amazon S3 (which can also connect to Walrus [12]), Google App Engine (plus the corresponding service running there, which is compatible to AppScale [13]), and for local harddisks. Further nodes are planned. In theory, there are no system limitations to extensibility.

Every node shares two message queues with its corresponding Distributor. Incoming messages are checked for their request type and then mapped to the respective methods which returns a response message. The methods provided by the nodes are GET, PUT, LISTFILES and DELETE of which each node assumes that they can be invoked multiple times simultaneously, i.e. all synchronization issues on this level are pushed to the underlying infrastructure services.

C. Distributor

The Distributor is situated in the second layer from the bottom and it is the component within MetaStorage which is "doing the actual work". The Distributor alone is responsible for replication and retrieval of files. All components on a higher layer are usually granted fragmentation transparency. Requests to the Distributor are also sent asynchronously for which purpose every Distributor holds an input and output queue. All operations offered by the Distributor are idempotent, so, if an error occurs one may just resend the request.

The Distributor implementation bases its distribution mechanism on an approach presented by DeCandia et al. [8] as well as Lakshman and Malik [14] which describes the concept of a preference list based on the hash of the key. Depending on the preferred Cloud storage services and their order within the preference list files are stored on the first N nodes and, thereby, distributed to multiple providers. Since MetaStorage is a quorum-based system [15] already R successful reads (and W for writes respectively) are sufficient to return success. Whereas in Dynamo the preference list originally contained physical and later on logical nodes we adopted and changed the approach to fit into our scenario: N, R and W can still be configured but the preference list is identical for the entire key range, which makes sense because an entire Cloud storage service is less likely to fail than a single machine and is also expected to have a load balancing scheme of its own. We, thus, have no need for partitioning algorithms like consistent hashing [16], [17]. So, every Distributor instance contains a preference list which is an ordered list of MetaStorage nodes. Changes to the preference list and the (N_P, R, W) configuration are also possible at runtime. Whenever this is done the system halts and waits until all active requests have terminated. Upon completion the changes are applied and all processes get restarted. Apart from removing the need of partitioning algorithms the static preference list also gives us the second knob to balance consistency-latency tradeoffs as we already pointed out.

There is one difference to [15], though: Quorum-based systems usually require a configuration where $R + W > N$ to avoid reading stale data as well as $W > \frac{N}{2}$ to avoid conflicts arising from concurrent writes. Since this also affects availability these requirements have been ignored in both Dynamo and MetaStorage.

In the following we will present the design of the GET and PUT operations of the Distributor. See also table I for a brief overview of all supported operations.

1) PUT: Whenever the Distributor receives a PUT request it rebroadcasts it to the first N nodes of the preference list. Afterwards, the Distributor waits for responses. Whenever a response is of type error a new PUT request is created and sent to the next node of the preference list which has so far not

TABLE I
OPERATIONS

Operation	Functionality	Terminate	Response Type	Parameters
DELETE	BR[b] to all nodes	Instantly	ACK[a]	Key
ASSERTEDDELETE	BR[b] to all nodes	Upon receipt of responses or after timeout	ACK[a] or list of failed nodes	Key
GET	BR[b] to all nodes storing replica	Upon receipt of R identical responses, N responses or after timeout	Success if more than R identical responses, else failure. Includes all retrieved data.	Key
PUT	BR[b] to first N nodes, re-broadcast until N nodes store a replica	Upon receipt of W ACKs[a] or if the preference list contains not enough nodes	Success if at least W replica exist, else failure.	Key, Value
LISTFILES-LT†	BR[b] to all nodes	Upon receipt of responses or after timeout	List of keys with corresponding number of available replica	None
LISTFILES	BR[b] to all nodes	Upon receipt of responses or after timeout	List of keys per node sorted by nodes	None

† LT = location transparency [a] ACK = acknowledgement [b] BR = broadcast requests

been contacted. As soon as W nodes have returned a success message the PUT operation terminates and responds to the requester. But while $W < N_P$ the system continues in the background to bring up the number of replica from W to N. In any case, if less than W nodes respond with success and every node has already been contacted, an error message is returned.

When the file has finally been stored on N nodes the system checks whether those N nodes are identical to the first N nodes of the preference list. If not so-called Hinted Handoffs [8] are created and kept locally in memory. A Hinted Handoff contains three pieces of information: The node of the first N nodes of the preference list which reported an error, the node which stored the file instead and the affected file key.

The Distributor includes several subprocesses which periodically try to resolve the existing Hinted Handoffs. Details are beyond the scope of this paper.

There is one special case for which we have not been able to find a solution so far: If W nodes acknowledge storing the data but all other nodes in the preference list fail, a success message has already been returned because the algorithm could not know in advance that it would not be possible to bring the number of replica up to N. So far, there will not be more than W replica until the point where more nodes are available again and another GET or a PUT request is issued. To reduce the chance of such a situation occurring we propose sufficiently long preference lists combined with a few local file system nodes to cache the data in between.

2) GET: Whenever GET is invoked the operation retrieves the preference list and queries the list of Hinted Handoffs. Based on the request's key an updated temporary preference list is created which contains all N nodes which hold a copy of the requested file. Future versions might query the first R healthy nodes in the absence of Hinted Handoffs. This would allow a lazier synchronisation with other Distributor instances. Next, messages containing GET requests for the respective key are sent to all nodes on the temporary preference list. Afterwards, the Distributor waits for the node's responses. There are several cases:

1) If the response messages contain R identical payloads then that payload is returned. For any error amongst the remaining $(N_P - R)$ responses a PUT request with the majority content is issued to the respective node to increase durability (read repair mechanism).
2) If the first R responses contain different payloads the system waits until it has received N responses and then responds with a success message containing all retrieved file versions.
3) If less than R nodes return the requested file the method responds with an error message including all retrieved versions if any exist.

In all cases but the first one the application should preferrably determine the correct version and write it back to resolve the pending conflicts or durability issues.

3) Further Operations: Apart from GET and PUT Meta-Storage also provides two operations to list all stored files (comparable to the Linux command ls or the DOS command dir) as well as to delete specific files. We propose to choose one of the two versions based on the specific usecase. For more information on all operations see table I. In section V-A we discuss the latency-consistency tradeoffs which can be addressed by choosing among the two delete operations DELETE and ASSERTEDDELETE. This small knob exists independent of the provider selection.

D. Coordinator

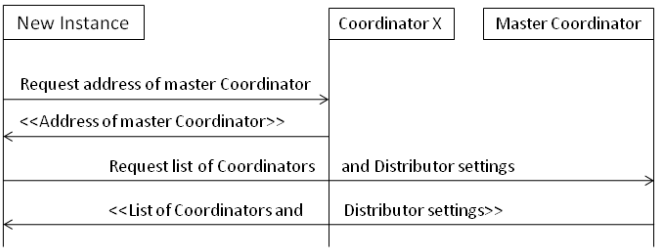

Fig. 2. Overview of Coordinator Bootstrapping

When we combine a Distributor instance with some nodes we already have a running system which processes incoming messages, evaluates and executes their requests and returns responses. There is one issue, though: We are in a highly scalable environment and every underlying storage infrastructure is

presumed to be scaling as well (Elson and Howell [18] reason why scalability is so much of importance). But if we use only one Distributor we will create a perfect bottleneck in our application landscape. To avoid this, we thought about adding independent Distributor instances but quickly discovered that some coordination between them is necessary. For example, every Distributor should have the same preference list and (N_P, R, W) configuration. Also, with every PUT request the set of Hinted Handoffs might change but other instances would not know about it. So, we finally added another layer on top of the Distributor: the Coordinator. Essentially, the task of our Coordinator is to manage the state of the underlying Distributors and to keep them all up-to-date in terms of configuration or membership changes. Figure 1 shows how MetaStorage becomes scalable by the use of Coordinators.

In order to avoid a fully centralized system but also, for ease of implementation, a completely decentralized system we propose a semi-decentralized solution: There is a master Coordinator which determines all other Coordinator's state. Now we had to cope with failing master Coordinators instead and solved this by giving every Coordinator a complete ordered list of all Coordinators within a system. Whenever the master cannot be reached for a certain period of time the remaining Coordinators each assume the master to be offline and remove it from their list of Coordinators (Lindsay [19] provides arguments in favor of local action in case of failures). Thus, the former No. 2 becomes the new No. 1 and master. Since Coordinators know about all other Coordinators and their specific order every one of them can decide – without central control – who the new master is as well as when it becomes a master.

New Coordinators are always appended to the list of Coordinators so that the list is ordered by the total length of server uptime. This guarantees that every Coordinator which knows of more than three Coordinators (the master, some other Coordinator and itself) always knows No. 2. So, this implies that – when the master fails – every Coordinator which was not only known to the master before it failed also knows about No. 2.

This leaves only two issues:

1) What happens if a Coordinator registers with the master but the master fails before it can respond?
2) What happens if a Coordinator registers with the master, the master responds but fails before it can forward the information on the novice to the other Coordinators?

Case (1) is simple: The novice cannot know about its Distributor's configuration and will simply shut down while the other Coordinators do not know about the novice and are thus not affected. We could then simply restart the novice. (2) Status quo is that the novice knows about the system but not the other way around. So, when the novice realizes that it cannot reach the system's master it will try to reach No. 2. That way No. 2 gets to know about the novice, appends the new Coordinator to the end of the list and responds with the new list. The novice overwrites its own list and we again have consistent information on all sides.

Figure 2 shows how bootstrapping of new Coordinators works. Coordinator X, in the case of the diagram, may be any existing Coordinator - even the master itself.

Since MetaStorage requires only a very limited amount of coordination (we presume configuration changes to be rare) this very simple protocol suffices for our purposes. Other approaches, including Paxos-like algorithms [20], [21], quorum-like systems [22] or the Google approach based on Chubby [23] which allows master election through locking, were considered but deemed too complicated to implement while providing more coordination than necessary for our purposes.

Regarding partitioning tolerance there should not be much of a problem: All MetaStorage instances are more or less independent so that there are only a few implications if the system is split into two or more subsystems:

1) Hinted Handoffs are no longer forwarded to other subsystems and there might be several subsystems trying to resolve them.
2) It is necessary to manually merge the subsystems again, once the partitioning has been resolved.
3) Changes in configuration need to be issued to all subsystems individually or reissued once the partitioning has been resolved.

Apart from these three points there should not be any further implications beyond side effects (e.g. slightly reduced availability during GETs caused by a lack of knowledge on Hinted Handoffs; this problem could be addressed by systems querying more than the first N nodes during GETs as already noted earlier).

The Coordinator class also contains some subcomponents of which especially the UpdateManager is of importance as it periodically creates a consistent view of the system's state but is only running for non-master Coordinators. For this purpose, it creates every ten seconds (customizable) a message containing information on the list of Hinted Handoffs held by its Distributor as well as the timestamp of its last update to the list of Coordinators and (N_P, R, W) configuration. It then sends this message to the master Coordinator which compares the timestamp to its own Coordinator list's timestamp and creates a response message. If the timestamps differ the master appends his own list of Coordinators and (N_P, R, W) settings to the message. Afterwards, the master creates a merged version of the Hinted Handoffs based on the file key and the individual Hinted Handoff timestamp. Resolved Hinted Handoffs are thus excluded. This merged version is also appended to the message which is then sent back to the requester. The recipient overwrites its Hinted Handoff list as well as the list of Coordinators and settings if necessary and updates the respective timestamp.

Now both parties, the master and the requester, share the same view on the situation. This mechanism guarantees that under non-adverse conditions every Coordinator should be up-to-date after a maximum of twice the update interval, i.e. with current standard settings after twenty seconds.

E. MetaStorageHost

Surrounding the Coordinator there is an entire collection of utility classes or functions. One of the most useful ones is the MetaStorageHost. Basically, it is a local registry for local MetaStorage instances and, hence, allows to run more than one Coordinator-Distributor pair within the same Java Virtual Machine. This could be useful to fully take advantage of machines with lots of CPU cores. Since all instances are identified by unique IDs a MetaStorageHost can forward incoming requests to the specific instance associated with the ID.

Apart from its function as a registry the MetaStorageHost is also responsible for information and functionality shared by all Coordinators running within the same Java VM. This includes hosting the Web Service interfaces as well as handling all incoming and outgoing requests for which it also provides parameter transformations, syntax checks and authentication. Furthermore, the host includes message handlers to map from synchronous SOAP requests to asynchronous internal messaging. Future versions might also allow asynchronous SOAP requests with callbacks.

F. Security

Security measures in MetaStorage include a role-based user management which allows to distinguish between different rights as well as several security levels with the corresponding demands on the system and (as a future extension) the option to enable encryption before persisting data in the Cloud.

While some nodes already communicate via https every single Web Service call is still unencrypted. This is due to limitations of the used JAX-WS implementation which only supports http. Of course, this critically affects security so that we plan to include another JAX-WS server implementation in future versions. Another aspect is file encryption: Right now, many enterprises avoid (public) Cloud offerings as internal guidelines forbid storing internal data off-premises. To offer MetaStorage also in this context it could easily be achieved that every file passing MetaStorage is encrypted before writing it to the Cloud, i.e. before it leaves the responsibility of the customer. The latter approach is also taken in other systems which are "paranoid" in the sense that they consider their storage nodes to be an, at least potentially, hostile environment. Examples include Farsite [24], HAIL [25], Oceanstore [26] or Antiquity [22].

III. Evaluation of MetaStorage

MetaStorage is an eventually consistent, fully replicating distributed storage system layered on top of multiple Cloud storage services. In the following, we describe our test setup to measure the length of inconsistency windows during PUT requests as well as latency overheads. Temporary inconsistencies are caused by update requests, such as PUT, DELETE, and ASSERTEDDELETE, and persist for the time period between updating the first and the Nth replica.

Our first test setup for measuring inconsistency windows is MetaStorage with a (3,1,1) Quorum configuration using only local file system nodes. The workload is set to 1,000 PUT requests, each writing 100 Bytes which we repeat four times.

We choose this configuration for two reasons:

1) A (3,1,1) configuration asserts that one stale replica is sufficient to return inconsistent (i.e. stale) results. In this configuration, it is important to estimate the inconsistency window. Furthermore, a number of three replica is widely used in replicated storage systems [8].
2) We choose to store files locally since this allows us to efficiently measure the exact moment in time when a file is written. Also, it removes any issues arising from clock synchronization of several machines. As the time necessary to store such a small file on local harddisk is negligible these results can without loss of generality be extended to any other usecase by simply adding the overhead of issuing a Web Service call to a remote storage service.

We measure the time between invoking a PUT request and receiving a response message, as well as the time between receiving a response message from the first replica and the time necessary to update the other two replica. As our algorithm has to actively poll the file system, our results for the inconsistency windows are pessimistic.

We observe that the MetaStorage system layer induces fairly stationary inconsistency windows of 0.09ms which are negligible.

Our second test setup for measuring the latency overhead of MetaStorage is a (3,2,2) Quorum configuration with the following preference list entries: an S3 node, a Google App Engine node and three local file system nodes. The test setup is installed on an Amazon EC2 large instance while our test client runs on a small instance, both in the availability region US-East. All latency tests were executed in mid-June, 2010 during a period when Google App Engine was experiencing huge problems with datastore latency and timeouts [27]. Since we do not want to run load testing, our algorithm issues only one PUT request at a time. The test workload is set to 10,000 times at 100 Byte and 5,000 times at 1 KB, 10 KB and 100 KB each.

Due to the availability problems of Google App Engine during the test period, out of a total of 50,000 requests to Google App Engine 11,204 (more than 22%) returned an http code 500 or 503. Amazon S3, in contrast, produced only two errors (0.004%). Nevertheless, MetaStorage was able to serve all requests with 100% availability despite the degraded availability of the underlying Cloud storage services. Without MetaStorage's Hinted Handoffs, the aggregate availability would have been 77.6%.

We experienced high volatility of latency measures, mainly due to availability problems combined with missing timeouts. Therefore, we analyze the data by calculating the median which is more robust towards outliers and skewed distributions than other statistical figures. In our tests, the latency overhead of MetaStorage amounts to approximately 300ms.

IV. Related Work

There is preliminary work on distributed storage systems to overcome vendor lock-in and to improve availability of stored data. Bunch et al. [28] extended the AppScale platform with unified access to diverse Cloud storage services using the Google App Engine Storage API. AppScale, however, can only connect to one data store and applications deployed on the platform are restricted to this connection.

Broberg et al. [29], [30] leverage multiple Cloud storage systems to increase the performance of content delivery with a Meta Content Delivery Network (MetaCDN) and developed a prototype to evaluate performance gains of their approach. MetaCDN focuses on read performance needed for fast content delivery and, therefore, replicates data to many Cloud storage services. To improve reads MetaCDN routes each content request to the replica available with the lowest expected latency. MetaCDN, however, lacks support for adequate write performance and immediate replication and, thus, cannot be employed as a full-fledged storage system.

Similarly, Bowers et al. [25] developed a High Availability and Integration Layer (HAIL) that stores data in encrypted files distributed over multiple storage services and returns decrypted data upon read requests with low compute effort. HAIL improves data security by utilizing encryption and data distribution over multiple Cloud storages but disregards scalability and introduces a bottleneck as it excludes a component comparable to our Coordinator.

With Redundant Array of Cloud Storage (RACS) Abu Libdeh et al. [31] propose a Cloud storage overlay system which acts as a proxy that uses erasure coding [32] to distribute files over multiple Cloud storages, simulating a Redundant Array of Independent Disks (RAID) system. However, every write operation terminates only when all Cloud storage services have completed the operation, leading to high latencies for data that is distributed world-wide. Furthermore, as RACS is not based on full replication it requires huge numbers of storage offerings which might not even exist in the first place. Also, built on top of eventually consistent [33] storage services RACS might fail in retrieving any data at all while other systems should at least return an outdated version.

Brantner et al. [34], [35] present a database system that builds on Amazon's S3 Cloud storage and, thereby, is tied to a single storage service. Future enhancements of the approach might include support for multiple Cloud storage services, but the effects on this database approach are not clear and not evaluated, yet.

Unlike existing approaches MetaStorage aims at being a versatile, distributed meta Cloud storage service that leverages the diverse capabilities of existing Cloud storage services and also considers consistency guarantees of its underlying storage services. With an implementation of Dynamo's quorum protocol [8] MetaStorage further resolves consistency conflicts during read operations and supports consistency rationing over Cloud storages via (N_P, R, W) consistency levels [15], [36]. By communicating consistency problems openly MetaStorage allows conflict resolution mechanisms at application level (see also [37]–[39]).

V. Discussion

MetaStorage offers three major advantages over using a single Cloud storage service. First, MetaStorage reduces vendor lock-in by distributing data across the infrastructure of different Cloud storage vendors. Second, services on top of MetaStorage can improve availability, durability and client latency by replicating their data across multiple Cloud storage services. Third, MetaStorage enables storage service differentiation by selecting and prioritizing Cloud storage services in the preference list.

There are some problems which MetaStorage cannot fix: services on top of MetaStorage inherit weak consistency guarantees of the underlying Cloud storage services. However, MetaStorage exposes these consistency guarantees to the upper service layer and thereby enables application-layer conflict resolution mechanisms.

MetaStorage also introduces drawbacks compared to using a single Cloud storage service. First of all, MetaStorage causes monetary overhead due to redundant data storage. Second, MetaStorage causes a latency overhead because it adds an additional layer to the system stack that must be traversed for every service invocation. Third, MetaStorage relaxes consistency guarantees by Dynamo-style replication protocols.

The monetary overhead depends on the pricing scheme of the underlying Cloud storage services. The minimum latency overhead for each service request is the time necessary to issue a MetaStorage Web Service call plus the overhead imposed by the replication protocol. Problems related to the weaker consistency guarantees of MetaStorage and the underlying Cloud storage services arise as a consequence of update requests. However, different from other distributed storage systems, MetaStorage offers additional configuration knobs to tweak consistency and latency guarantees as well as other quality of service attributes at runtime.

Two mechanisms enable MetaStorage to offer differentiated Cloud storage services by configuring high-level consistency guarantees at runtime. The first mechanism is the MetaStorage preference list of Cloud storage services; the second mechanism is the Sloppy Quorum replication protocol with Hinted Handoffs, (N_P, R, W). For example, the preference list allows tuning data storage towards low latency by moving low-latency Cloud storage services to the top of the list; it allows highly available and durable services by moving highly redundant Cloud storage services, such as Amazon S3, to the top of the list. The second mechanism, Sloppy Quorum configurations of parameters (N_P, R, W) can be used to tune MetaStorage towards write-optimized or read-optimized data access. These two mechanisms can be used in various combinations to offer a wide range of services qualities.

A. Latency versus Consistency Tradeoffs

Applications on top of MetaStorage can specify their optimal solution of tradeoff decisions between low latency and

specific consistency guarantees. MetaStorage offers a plurality of mechanisms for this purpose.

The time period of inconsistency caused by write operations can be estimated by this simple equation:

$$t_{inc} = t_{max} - t_{min} + t_{ms}$$

where t_{inc} is the size of the inconsistency window, t_{max} is the longest and t_{min} the shortest time period that is necessary to store a file on a Cloud storage service. Our evaluation has shown that the inconsistency window induced by MetaStorage, t_{ms}, is negligible.

The equation shows that the latency difference between Cloud services in the preference list affects consistency guarantees. While one fast service might be good for total MetaStorage latency, it is bad regarding consistency guarantees as the length of the inconsistency window only depends on the latency difference between the fastest and the slowest storage service. MetaStorage here trades consistency against latency and not against availability – a different tradeoff as stated by the CAP theorem [40].

Nevertheless, if Hinted Handoffs need to be created, t_{inc} increases by the time that is necessary for one of the first N_P nodes to return its error message minus t_{min}. Furthermore, the participating set of storage services considered for our calculation changes which might possibly affect t_{max} and/or t_{min}.

The inconsistency caused by delete operations can be tweaked, as well. Using DELETE instead of ASSERTEDDELETE, decreases latency at the cost of consistency. For example, in a (3,2,W) configuration, there are four possible results:

1) All nodes report success.
2) One node reports a failure.
3) Two nodes report a failure.
4) Three nodes report a failure.

Cases 3 and 4 are very unlikely to occur. However, should they really happen, the requester will learn about the failure and can resolve the problem. The simplest resolution would be to reissue the request or as an alternative just to keep a list of "forbidden keys" so that, whenever a GET request is received for this particular key, it is denied until a PUT request for the specific key has returned success.

Even if an application cannot afford the performance downsides of ASSERTEDDELETE but nevertheless requires stronger guarantees that a file has really been removed one can still twist the scenario by setting the (N_P, R, W) configuration which also affects consistency as well as latency.

B. Tuning Consistency Guarantees

For the purpose of maximizing MetaStorage consistency guarantees without seriously affecting availability, one needs to minimize the length of inconsistency windows, i.e. the duration between R replica being written and N replica being written. The optimal way to reach this goal is by having equally fast storage services as the first N_P preference list entries. If that is not possible, as a second-best solution, the first N nodes should be ordered by their write latency with the first node being the slowest and the Nth node being the fastest. As the necessity of one Hinted Handoff is more likely than multiple Hinted Handoffs, the $(N_P + 1)$th node should, for efficiency reasons, be the fastest storage service. Again, for efficiency reasons the positions larger than $N_P + 1$ should increase regarding latency.

C. Tuning Latency Guarantees

Another conclusion we draw from these results is the importance of short timeouts: Since MetaStorage follows the Hinted Handoff approach [8] a timeout will most likely not affect availability. Still, it would heavily improve latency for requests in situations with lots of errors where files could be cached, e.g. locally, and later on written to a remote site using the HintedHandoffResolver. This also corresponds to the basic idea that in a distributed system local proximity and/or high-bandwidth connections are advantageous towards latency.

VI. Conclusion

In this paper, we presented the design and implementation of the MetaStorage system, a federated architecture that utilizes diverse Cloud storage providers. MetaStorage implements a replication scheme based on Amazon's Dynamo, but elevates concepts to a network of (autonomous and heterogeneous) storage providers. We have shown that MetaStorage increases overall availability compared to any individual provider. Furthermore, MetaStorage introduces provider configurations (in preference lists) as a new means beyond existing configurations of traditional quorum systems and thus provides additional control mechanisms to manage consistency-latency tradeoffs.

There are still a number of open questions, though, which we plan to address in our future work:

- What tradeoffs do exist beyond CAP or consistency-latency?
- How are the different tradeoffs interrelated?
- How can we measure consistency or any other relevant property?
- What are the scalability limitations of MetaStorage?
- Can we derive a method for the optimal selection of parameters (N_P, R, W)?
- What is the relation between parameter configurations and the corresponding output in terms of consistency, latency, availability etc.?
- Is it possible to extend the set of MetaStorage operations without sacrificing compatibility?

We believe that a system like MetaStorage provides an ideal testbed to address these questions, as both controlled environments (using local nodes only) as well as federated environments (using autonomous service providers) can be tested. Our current research uses the testbed to answer the question "How soon is eventual?" in the context of eventual consistency.

ACKNOWLEDGMENT

The work described in this paper was fully supported by the German Federal Ministry of Education and Research (BMBF) under grant 01IC10S01A.

REFERENCES

[1] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm, "What's inside the Cloud? An architectural map of the Cloud landscape," in *Software Engineering Challenges of Cloud Computing, 2009. CLOUD'09. ICSE Workshop on*. IEEE, 2009, pp. 23–31.

[2] C. Baun, M. Kunze, J. Nimis, and S. Tai, *Cloud Computing: Web-basierte dynamische IT-Services*, ser. Informatik im Fokus. Berlin: Springer, 2010.

[3] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, H. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the Clouds: A Berkeley View of Cloud Computing." UCB/EECS-2009-28, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2009.

[4] J. Brodkin, *More Outages hit Amazon's S3 Storage Service*. Network World, Jul. 2008, (accessed on October 19, 2010). [Online]. Available: http://www.networkworld.com/news/2008/072108-amazon-outages .html

[5] R. Cellan-Jones, *The Sidekick Cloud Disaster*. BBC, Oct. 2009, (accessed on October 19, 2010). [Online]. Available: http://www.bbc.co.uk/blogs/technology/2009/10/the_sidekick_ cloud_disaster.html

[6] D. Ionescu, *Microsoft Red-Faced After Massive Sidekick Data Loss*. PC World, Oct. 2009, (accessed on October 19, 2010). [Online]. Available: http://www.pcworld.com/article/173470/microsoft_redfaced_ after _massive_sidekick_data_loss.html

[7] NetworkWorld, *From Sidekick to Gmail: A Short History of Cloud Computing Outages*. Network World, Oct. 2009, (accessed on October 19, 2010). [Online]. Available: http://www.networkworld.com/news/2009/101209-sidekick-cloud-computing-outages-short-history.html

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proc. SOSP*, 2007.

[9] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned, scalable Internet services," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 230–243, 2001.

[10] S. Garfinkel, "An Evaluation of Amazon's Grid Computing Services: EC2, S3, and SQS," in *Center for*. Citeseer, 2007.

[11] A. T. Velte, T. J. Velte, and R. Elsenpeter, *Cloud Computing: A Practical Approach*. Upper Saddle River, NJ: McGraw-Hill, 2010.

[12] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, 2009, pp. 124–131.

[13] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski, "Appscale: Scalable and open appengine application development and deployment," *First International Conference on Cloud Computing*, 2009.

[14] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[15] R. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 2, pp. 180–209, 1979.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997, pp. 654–663.

[17] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2001, pp. 149–160.

[18] J. Elson and J. Howell, "Handling flash crowds from your garage," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. USENIX Association, 2008, pp. 171–184.

[19] S. Bourne, "A conversation with Bruce Lindsay," *Queue*, vol. 2, no. 8, pp. 22–33, 2004.

[20] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007, pp. 398–407.

[21] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[22] H. Weatherspoon, P. Eaton, B. Chun, and J. Kubiatowicz, "Antiquity: exploiting a secure log for wide-area distributed storage," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 371–384, 2007.

[23] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 335–350.

[24] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer, "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 1–14, 2002.

[25] K. Bowers, A. Juels, and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 187–198.

[26] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells *et al.*, "Oceanstore: An architecture for global-scale persistent storage," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 5, pp. 190–201, 2000.

[27] Google, *Datastore Performance Growing Pains*. Google, Jun. 2010, (accessed on December 4, 2010). [Online]. Available: http://googleappengine.blogspot.com/2010/06/datastore-perfor mance-growing-pains.html

[28] C. Bunch, N. Chohan, C. Krintz, J. Chohan, J. Kupferman, P. Lakhina, Y. Li, and Y. Nomura, "An evaluation of distributed datastores using the appscale cloud platform," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, 2010, pp. 305 –312.

[29] Broberg, Buyya, and Tari, "Creating a Cloud Storage Mashup for High Performance, Low Cost Content Delivery," in *Service-Oriented Computing–ICSOC 2008 Workshops*. Springer, 2009, pp. 178–183.

[30] J. Broberg, R. Buyya, and Z. Tari, "MetaCDN: Harnessing 'Storage Clouds' for high performance content delivery," *Journal of Network and Computer Applications*, vol. 32, no. 5, pp. 1012–1022, 2009.

[31] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: a case for cloud storage diversity," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 229–240.

[32] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," *Peer-to-Peer Systems*, pp. 328–337, 2002.

[33] W. Vogels, *Eventually Consistent - Revisited*, Dec. 2008, (accessed on October 19, 2010). [Online]. Available: http://www.allthingsdistributed.com/2008/12/eventually _consistent.html

[34] Brantner, Florescu, Graf, Kossmann, and Kraska, "Building a database on S3," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 251–264.

[35] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a Database in the Cloud." ETH Zürich, 2009.

[36] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency Rationing in the Cloud: Pay only when it matters," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 253–264, 2009.

[37] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek, "Resolving file conflicts in the Ficus file system," in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference-Volume 1*. USENIX Association, 1994, p. 12.

[38] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on computers*, pp. 447–459, 1990.

[39] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 172–182, 1995.

[40] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, p. 59, 2002.