Complying with Data Handling Requirements in Cloud Storage Systems

Martin Henze, Roman Matzutt, Jens Hiller, Erik Mühmer,
Jan Henrik Ziegeldorf, Johannes van der Giet, and Klaus Wehrle

Abstract—In past years, cloud storage systems saw an enormous rise in usage. However, despite their popularity and importance as underlying infrastructure for more complex cloud services, today's cloud storage systems do not account for compliance with regulatory, organizational, or contractual data handling requirements by design. Since legislation increasingly responds to rising data protection and privacy concerns, complying with data handling requirements becomes a crucial property for cloud storage systems. We present PRADA, a practical approach to account for compliance with data handling requirements in key-value based cloud storage systems. To achieve this goal, PRADA introduces a transparent data handling layer, which empowers clients to request specific data handling requirements and enables operators of cloud storage systems to comply with them. We implement PRADA on top of the distributed database Cassandra and show in our evaluation that complying with data handling requirements in cloud storage systems is practical in real-world cloud deployments as used for microblogging, data sharing in the Internet of Things, and distributed email storage.

Index Terms—cloud computing, data handling, compliance, distributed databases, privacy, public policy issues

1 Introduction

Nowadays, many web services outsource the storage of data to cloud storage systems. While this offers multiple benefits, clients and lawmakers frequently insist that storage providers comply with different data handling requirements (DHRs), ranging from restricted storage locations or durations [1], [2] to properties of the storage medium such as full disk encryption [3], [4]. However, cloud storage systems do not support compliance with DHRs today. Instead, the selection of storage nodes is primarily optimized towards reliability, availability, and performance, and thus mostly ignores the demand for DHRs. Even worse, DHRs are becoming increasingly diverse, detailed, and difficult to check and enforce [5], while cloud storage systems are becoming more versatile, spanning different continents [6] or infrastructures [7], and even second-level providers [8]. Hence, clients cannot ensure compliance with DHRs when their data is outsourced to cloud storage systems.

This apparent lack of control is not merely an academic problem. Since customers have no influence on the treatment of their data in today's cloud storage systems, a large set of customers cannot benefit from the advantages offered by the cloud. The Intel IT Center surveys [9] among 800 IT professionals, that 78% of organizations have to comply with regulatory mandates. Again, 78% of organizations are concerned that cloud offers are unable to meet their requirements. In consequence, 57% of organizations actually refrain from outsourcing regulated data to the cloud. The lack of control over the treatment of data in cloud storage hence scares away many clients. This especially holds for the healthcare, financial, and government sectors [9].

Supporting DHRs enables these clients to dictate adequate treatment of their data and thus allows cloud storage operators to enter new markets. Additionally, it empowers operators to efficiently handle differences in regulations [10] (e.g., data protection). Although the demand for DHRs is widely acknowledged, practical support is still severely limited [9], [11], [12]. Related work primarily focuses on DHRs while processing data [13], [14], [15], limits itself to location requirements [16], [17], or treats the storage system as a black box and tries to coarsely enforce DHRs from the outside [12], [18], [19]. Practical solutions for supporting arbitrary DHRs when storing data in cloud storage systems are still missing – a situation that is disadvantageous to clients and operators of cloud storage systems.

Our contributions. In this paper, we present PRADA, a general key-value based cloud storage system that offers rich and practical support for DHRs to overcome current compliance limitations. Our core idea is to add one layer of indirection, which flexibly and efficiently routes data to storage nodes according to the imposed DHRs. We demonstrate this approach along classical key-value stores, while our approach also generalizes to more advanced storage systems. Specifically, we make the following contributions:

1) We comprehensively *analyze* DHRs and the challenges they impose on cloud storage systems. Our analysis shows that a wide range of DHRs exist, which clients and operators of cloud storage systems have to address.
2) We present PRADA, our approach for *supporting DHRs in cloud storage systems*. PRADA adds an indirection layer on top of the cloud storage system to store data tagged with DHRs only on nodes that fulfill these

- *M. Henze is with the Fraunhofer Institute for Communication, Information Processing and Ergonomics FKIE, Germany.*
E-mail: martin.henze@fkie.fraunhofer.de
- *R. Matzutt, J. Hiller, J. H. Ziegeldorf, J. van der Giet, and K. Wehrle are with the Chair of Communication and Distributed Systems at RWTH Aachen University.*
E-mail: {matzutt, hiller, ziegeldorf, giet, wehrle}@comsys.rwth-aachen.de
- *E. Mühmer is with the Chair of Operations Research at RWTH Aachen University, Germany.*
E-mail: muehmer@or.rwth-aachen.de

Manuscript submitted June 19, 2017. Revision submitted October 31, 2019.

requirements. Our design of PRADA is *incremental*, i.e., it does not impair data without DHRs. PRADA supports all DHRs that can be expressed as properties of storage nodes as well as any combination thereof. As we show, this covers a wide range of actual use cases.

3) We prove the *feasibility* of PRADA by implementing it for the distributed database Cassandra (we make our implementation available [20]) and by quantifying the costs of supporting DHRs in cloud storage systems. Additionally, we show PRADA's *applicability* in a cloud deployment along three real-world use cases: a Twitter clone storing two million authentic tweets, a distributed email store handling half a million emails, and an IoT platform persisting 1.8 million IoT messages.

A preliminary version of this paper appears in the proceedings of IEEE IC2E 2017 [21]. We extend and improve on our previous work in the following ways: First, we provide a detailed analysis and definition of the challenge of DHR compliance in cloud storage systems. Second, we extend PRADA with mechanisms for failure recovery. Third, we provide details on our implementation of PRADA. Fourth, we show the applicability of PRADA by realizing compliance with DHRs in three real-world use cases: a microblogging system, a distributed email system, and an IoT platform. Finally, we cover a broader range of related work and provide more detail on design, implementation, and evaluation.

Paper structure. In Section 2, we analyze DHRs and derive goals for supporting DHRs in cloud storage systems. We provide an overview of our design in Section 3, before we provide details on individual storage operations (Section 4), replication (Section 5), load balancing (Section 6), and failure recovery (Section 7). Subsequently, we describe our implementation in Section 8 and evaluate its performance and applicability in Section 9. We present related work in Section 10 and conclude with a discussion in Section 11.

2 DATA COMPLIANCE IN CLOUD STORAGE

With the increasing demand for sharing data and storing it at external parties [22], obeying with DHRs becomes a crucial challenge for cloud storage systems [11], [12], [23]. To substantiate this claim, we outline our setting and rigorously analyze existing and potentially future DHRs. Based on this, we derive goals that must be reached to adequately support DHRs in cloud storage systems.

2.1 Setting

We tackle the challenge of supporting DHR compliance in cloud storage systems which are realized over a set of nodes in different data centers [24]. To explain our approach in a simple yet general setting, we assume that data is addressed by a distinct key, i.e., a unique identifier for each data item. Key-value based cloud storage systems [25], [26], [27] provide a general, good starting point, since they are widely used and their underlying principles have been adopted in more advanced cloud storage systems [28], [29], [30]. We discuss how our approach can be applied to other types of cloud storage systems in Section 11.

As a basis for our discussion, we illustrate our setting in Figure 1. Clients (end users and companies) insert data into the cloud storage system and annotate it with DHRs. These requirements are in textual form and can be interpreted by the operator of the cloud storage system. The process of annotating data with DHRs is also known as sticky policies [31], [32] or data handling annotations [11], [23]. Each client of the storage system might impose individual and varying DHRs for each single inserted data item.

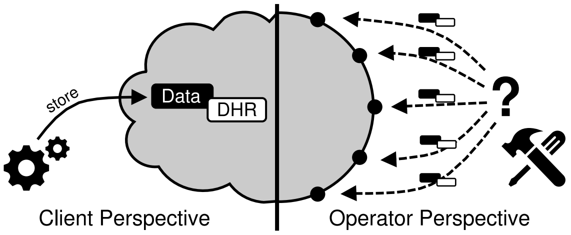


Fig. 1. **Setting.** When clients insert data with DHRs, the operator has to store it only on nodes of the storage system complying with the DHRs.

Compliance with DHRs has to be realized by the operator of the cloud storage system. Only the operator knows about the characteristics of the storage nodes and can thus make the ultimate decision on which node to store a specific data item. Different works exist that propose cryptographic guarantees [14], accountability mechanisms [33], information flow control [5], [34], or virtual proofs of physical reality [35] to relax trust assumptions on the operator, i.e., providing the client with *assurance* that DHRs are (strictly) adhered to. Our goals are different: Our main aim is for *functional* improvements of the status quo. Thus, these works are orthogonal to our approach and can possibly be combined if the operator is not sufficiently trusted.

2.2 Data Handling Requirements

We analyze DHRs from client and operator perspective and identify common classes, as well as the need to support also future and unforeseen requirements.

Client perspective. DHRs involve constraints on the storage, processing, distribution, and deletion of data in cloud storage. These constraints follow from legal (laws and regulations) [36], [37], contractual (standards and specifications) [38], or intrinsic requirements (user's or company's individual privacy requirements) [39], [40]. Especially for businesses, compliance with legal and contractual obligations is important to avoid serious (financial) consequences [41].

Location requirements relate to the storage location of data. On one hand, these requirements address concerns raised when data is stored outside of specified legislative boundaries [2], [11]. The EU's General Data Protection Regulation [37], e.g., forbids the storage of personal data in jurisdictions with an insufficient level of privacy protection. Also other legislation, besides data protection law, can impose restrictions on the storage location. German tax legislation, e.g., forbids the storage of tax data outside of the EU [23]. On the other hand, clients, especially corporations, can impose location requirements. To increase robustness against outages, a company might demand to store replicas of their data on different continents [39]. Furthermore, an enterprise could require that sensitive data is not stored at a competitor for fear of accidental leaks or deliberate breaches [40].

Duration requirements impose restrictions on the storage duration of data. The Sarbanes-Oxley Act (SOX) [42], e.g., requires accounting firms to retain records relevant to audits and reviews for seven years. Contrary, the Payment Card Industry Data Security Standard (PCI DSS) [38] limits the storage duration of cardholder data to the time necessary for business, legal, or regulatory purposes after which it has to be deleted. A similar approach, coined "the right to be forgotten", is actively being discussed and turned into legislation in the EU and Argentina [37], [43].

Traits requirements further define how data should be stored. For example, the US Health Insurance Portability and Accountability Act (HIPAA) [36] requires health data to be securely deleted before disposing or reusing a storage medium. Likewise, for the banking and financial services industry, the Gramm-Leach-Bliley Act (GLBA) [3] requires the proper encryption of customer data. Additionally, to protect against theft or seizure, clients may choose to store their data only on volatile [44] or fully encrypted [4] storage.

Operator perspective. The support of DHRs presents clear business incentives to cloud storage operators as it opens new markets and eases compliance with regulation.

Business incentives are given by the unique selling point that DHRs present to the untapped market of clients unable to outsource their data to cloud storage systems nowadays due to unfulfillable DHRs [9]. Indeed, cloud providers already adapted to some carefully selected requirements. To be able to sell its services to the US government, e.g., Google created the segregated "Google Apps for Government" and had it certified at the FISMA-Moderate level, which enables use by US federal agencies [41]. Furthermore, cloud providers open data centers around the world to address location requirements of clients [7]. From a different perspective, regional clouds, e.g., the envisioned "Europe-only" cloud [45], aim at increasing governance and control over data. Additionally, offering clients more control over their data reduces risks for loss of reputation and credibility [46].

Compliance with legislation is important for operators independent of specific business goals and incentives. As an example, the business associate agreement of HIPAA [36] requires the operator to comply with the same requirements as its clients when transmitting electronic health records [1]. Furthermore, the EU's General Data Protection Regulation [37] requires data controllers from outside the EU that process data originating from the EU to follow DHRs.

Future requirements. DHRs are likely to change and evolve just as legislation and technology are changing and evolving over time. Location requirements developed, e.g., since cloud storage systems began to span multiple geographic regions. As anticipating all possible future changes in DHRs is impossible, it is crucial that support for DHRs in cloud storage systems can easily adapt to new requirements.

Formalizing data handling requirements. To also support future requirements and storage architectures, we base our approach on a formalized understanding of DHRs that also covers yet unforeseen DHRs. To this end, we distinguish between different *types* of DHRs and consider different possible *properties* which storage nodes (can) support for a given type of DHRs. This makes it possible to compute the set of *eligible nodes* for a specified type of DHRs, i.e., those nodes that offer the properties requested by the client.

A simple example for a type of DHRs is *storage location*. In this example, the properties consist of all *possible storage locations*, and nodes whose storage location is equal to the one requested by the clients are considered eligible. In a more complicated example, we consider as DHR type the *security level of full-disk encryption*. Here, the properties range from 0 bits (no encryption) to different bits of security (e.g., 192 bits or 256 bits), with more bits of security offering a higher security level [47]. In this case, all storage nodes that provide at least the security level requested by the client are considered eligible to store the data.

By allowing clients to combine different types of DHRs and to specify a set of required properties (e.g., different storage locations) for each type, we provide them with powerful means to express DHRs. We detail how clients can combine different types in Section 4 and how we integrate DHRs into Cassandra's query language in Section 8.

2.3 Goals

Our analysis of real-world demands for DHRs based on legislation, business interests, and future trends emphasizes the importance to support DHRs in distributed cloud storage. We now derive a set of goals that any approach that addresses this challenging situation should fulfill:

Comprehensiveness: To address a wide range of DHRs, the approach should work with any DHRs that can be expressed as properties of storage nodes and support the combination of different DHRs. In particular, it should support the requirements in Section 2.2 and be able to adapt to new DHRs.

Minimal performance effort: Cloud storage systems are highly optimized and trimmed for performance. Thus, the impact of DHR support on the performance of a cloud storage system should be minimized.

Cluster balance: In existing cloud storage systems, the storage load of nodes can easily be balanced to increase performance. Despite having to respect DHRs (and thus limiting the set of possible storage nodes), the storage load of individual storage nodes should be kept balanced.

Coexistence: Not all data will be accompanied by DHRs. Hence, data without DHRs should not be impaired by supporting DHRs, i.e., it should be stored in the same way as in a traditional cloud storage system.

3 System Overview

The problem that has prevented support for DHRs so far stems from the common pattern used to address data in key-value based cloud storage systems: Data is addressed, and hence also partitioned (i.e., distributed to the nodes in the cluster), using a designated key. Yet, the *responsible node* (according to the key) for storing a data item will often not fulfill the client's DHRs. Thus, the challenge addressed in this paper is how to realize compliance with DHRs and still allow for key-based data access.

To tackle this challenge, the core idea of PRADA is to add an indirection layer on top of a cloud storage system. We illustrate how we integrate this layer into existing cloud storage systems in Figure 2. If a responsible node cannot comply with stated DHRs, we store the data at a different node, called *target node*. To enable the lookup of data, the responsible node stores a reference to the target for specific data. As shown in Figure 2, we introduce three new components (capability, relay, and target store) to realize PRADA.

Capability store: The global *capability store* is used to look up nodes that can comply with a specific DHR. Here, the operator of the cloud storage systems specifies for each node in the cluster which DHR properties this node can fulfill. To speed up lookups in the capability store, each node keeps a local copy of the complete capability store. This approach is feasible, as information on DHRs is comparably small and consists of rather static information. Depending on the individual cloud storage system, distributing this information can be realized by preconfiguring the capability store for a storage cluster or by utilizing the storage system itself for creating a globally replicated view of node capabilities. We consider all DHRs that describe static properties of a storage node and range from rather simplistic properties such as storage location to more advanced capabilities such as the support for deleting data at a specific date.

Relay store: Each node operates a local *relay store* containing references to data stored at other nodes. More precisely, it contains references to data the node itself is responsible for but does not comply with the DHRs. For each data item, the relay store contains the key of the data, a reference to the node at which the data is stored, and a copy of the DHRs.

Target store: Each node stores data that is redirected to it in a *target store*. The target store operates exactly as a traditional data store, but allows a node to distinguish data that falls under DHRs from data that does not.

Alternatives to adding an indirection layer are likely not viable for scalable key-value based cloud storage systems: Although it is possible to encode very short DHRs in the key used for data access [23], this requires knowledge about DHRs of a data item to compute the key for accessing it and disturbs load balancing. Alternatively, replication of all relay information on all nodes of a cluster allows nodes to derive relay information locally. This, however, severely impacts scalability of the cloud storage system and reduces the total storage amount to the limited storage space of single nodes.

Integrating PRADA into a cloud storage system requires us to adapt storage operations (e.g., creating and updating data) and to reconsider replication, load balancing, and failure recovery strategies in the presence of DHRs. In the following, we describe how we address these issues.

4 Cloud Storage Operations

The most important modifications of PRADA involve the CRUD (create, read, update, delete) operations. In the following, we describe how we integrate PRADA into the CRUD operations of our cloud storage model (cf. Section 2.1). We assume that queries are processed on behalf of the client by one of the nodes in the cluster, the *coordinator* node (as common in cloud storage systems [26]). Each node of the cluster can act as coordinator for a query and clients use the capability store to select a coordinator that complies with the requested DHRs. If no DHRs need to be considered, clients select a coordinator based on performance metrics such as proximity. For reasons of clarity, we postpone the discussion of the impact of different replication factors and load balancing decisions to Section 5 and 6, respectively.

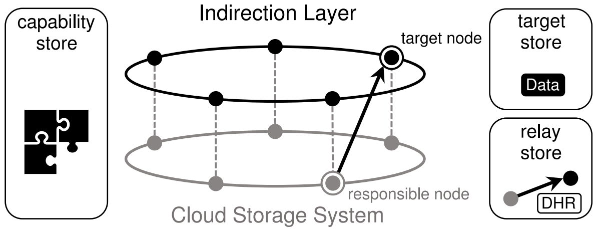
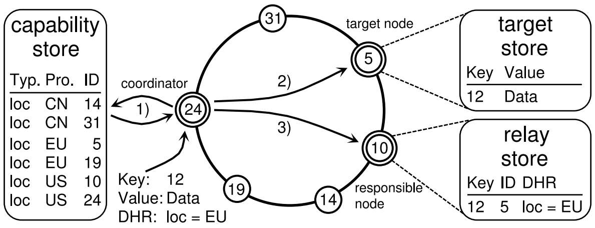


Fig. 2. **System overview.** PRADA adds an *indirection layer* to support DHRs. The *capability store* records which nodes support which DHR, the *relay store* contains references to indirected data, and the *target store* saves indirected data.



Typ.	Pro.	ID
loc	CN	14
loc	CN	31
loc	EU	5
loc	EU	19
loc	US	10
loc	US	24

Key	Value
12	Data

Key	ID	DHR
12	5	loc = EU

Fig. 3. **Creating data.** The coordinator derives nodes that comply with the DHRs from the capability store. It then stores the data at the target node and a reference to the data at the responsible node.

Create. The coordinator first checks whether a create request is accompanied by DHRs. If no requirements are specified, the coordinator uses the standard method of the cloud storage system to create data so that the performance of native create requests is not impaired. For all data *with* DHRs, a create request proceeds in three steps as illustrated in Figure 3. In Step 1, the coordinator derives the set of eligible nodes from the received DHRs, relying on the capability store (as introduced in Section 3) to identify nodes that fulfill all requested DHRs. Clients can combine different types of DHRs (e.g., location and support for deletion). Nodes are considered eligible if they support at least one of the specified properties for each requested type (e.g., one out of multiple permissible locations). Now, the coordinator knows which nodes of the cluster can comply with all requirements specified by the user and has to choose from the set of eligible nodes the target node on whom to store the data. It is important to select the target such that the overall storage load in the cluster remains balanced (we defer the discussion of this issue to Section 6). In Step 2, the coordinator forwards the data to the target, who stores it in its target store. Finally, in Step 3, the coordinator instructs the responsible node to store a reference to the actual storage location of the data to enable locating data upon read, update, and delete requests. The coordinator acknowledges the successful insertion after all three steps have been completed successfully. To speed up create operations, the second and third step—although logically separated—are performed in parallel.

Read. Processing read requests in PRADA is performed in three steps as illustrated in Figure 4. In Step 1, the coordinator uses the key supplied in the request to initiate a standard read query at the responsible node. If the responsible node does not store the data locally, it checks its local relay store for a reference to a different node. Should it hold such a reference, the responsible node forwards the read request (including information on how to reach the coordinator node for this request) to the target listed in the reference in Step 2. In Step 3, the target looks up the requested data in its target store and directly returns the query result to the coordinator. Upon receiving the result from the target, the coordinator processes the results in the same way as any other query result. If the responsible node stores the requested data locally (e.g., because it was stored without DHRs), it directly answers the request using the default method of the cloud storage system. In contrast, if the responsible node neither stores the data directly nor a reference to it, PRADA will report that no data was found using the standard mechanism of the cloud storage system.

Update. The update of already stored data involves the (potentially partial) update of stored data as well as the possible update of associated DHRs. In the scope of this paper, we define that DHRs of the update request supersede DHRs supplied with the create request and earlier updates. Other semantics, e.g., combining old and new DHRs, can be realized by slightly adapting the update procedure of PRADA. Consequently, we process update requests the same way as create requests (as it is often done in cloud storage systems). Whenever an update request needs to change the target node of stored data (due to changes in supplied DHRs), the responsible node has to update its relay store. Furthermore, the update request needs to be applied to the data (currently stored at the old target node). To this end, the responsible node instructs the old target node to move the data to the new target node. The new target node applies the update to the data, locally stores the result, and acknowledges the successful update to coordinator and responsible node and the responsible node updates the relay information. As updates for data without DHRs are directly sent to the responsible node, the performance of native requests is not impaired compared to an unmodified system.

Delete. Delete requests are processed analogously to read requests: The delete request is sent to the responsible node for the key that should be deleted. If the responsible node itself stores the data, it deletes the data as in an unmodified system. In contrast, if it only stores a reference to the data, it deletes the reference and forwards the delete request to the target. The target deletes the data and informs the coordinator about the successful deletion. We defer a discussion of recovering from delete failures to Section 7.

5 REPLICATION

Cloud storage systems employ replication to realize high availability and data durability [26]: Instead of storing a data item only on one node, it is stored on r nodes (typically, with a replication factor $1 \leq r \leq 3$). In key-value based storage systems, the r nodes are chosen based on the key of data (see Section 3). When complying with DHRs, we cannot use the same replication strategy. In the following, we thus detail how PRADA realizes replication instead.

Creating data. Instead of selecting only one target, the coordinator picks r targets out of the eligible nodes. The coordinator sends the data to all r targets and the list of all r targets to the r responsible nodes (according to the replication strategy of the cloud storage system). Consequently, each of the r responsible nodes knows about all r targets and can update its relay store accordingly.

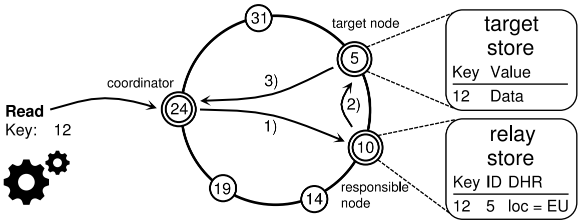


Fig. 4. **Reading data.** The coordinator contacts the responsible node to fetch the data. As the data was created with DHRs, the responsible node forwards the query to the target, which directly sends the response back to the coordinator.

Reading data. To process a read request, the coordinator forwards the read request to all responsible nodes. A responsible node that receives a read request for data it does not store locally looks up the targets in its relay store and forwards the read request to one of the r target nodes. To ensure that each target node receives a request, each responsible node uses the same consistent mapping between responsible and target nodes which is computed based on node identifiers. Each target that receives a read request sends the requested data to the coordinator for this request. If a read query is reissued due to a failure (cf. Section 7), each responsible node will forward the request to all r target nodes to increase reliability.

Impact on reliability. To successfully process a query in PRADA, it suffices if one responsible node and one target node are reachable. Thus, PRADA can tolerate the failure of up to $r-1$ responsible nodes and up to $r-1$ target nodes.

6 LOAD BALANCING

In cloud storage systems, load balancing aims to minimize (long term) load disparities in the storage cluster by distributing stored data and read requests equally among the nodes. Since PRADA drastically changes how data is assigned to and retrieved from nodes, existing load balancing schemes must be rethought. In the following, we describe a formal metric to measure load balance and then explain how PRADA builds a load-balanced storage cluster.

Load balance metric. Intuitively, a good load balancing aims at all nodes being (nearly) equally loaded, i.e., the imbalance between the load of nodes should be minimized. While underloaded nodes constitute a waste of resources, overloaded nodes drastically decrease the overall performance of the cloud storage system. We measure the load balance of a cloud storage system by normalizing the global standard deviation of the load with the mean load μ of all nodes [48]:

$$\mathfrak{L} := \frac{1}{\mu}\sqrt{\frac{\sum_{i=1}^{|N|}(\mathfrak{L}_i - \mu)^2}{|N|}}$$

with $\mathfrak{L}_i$ being the load of node $i \in N$. To achieve load balance, we need to minimize $\mathfrak{L}$. This metric especially penalizes outliers with extremely low or high loads, following the intuition of a good load balance.

Load balancing in PRADA. Key-value based cloud storage systems achieve a reasonably balanced load in two steps: (i) Equal distribution of data at insert time, e.g., by applying a hash function to identifier keys, and (ii) re-balancing the cluster if absolutely necessary by moving data between nodes. More advanced systems support additional mechanisms, e.g., load balancing over geographical regions [28]. Since our focus in this paper lies on proving the general feasibility of supporting data compliance in cloud storage, we focus on the properties of key-value based storage.

Re-balancing a cluster by moving data between nodes can be handled by PRADA similarly to moving data in case of node failures (Section 7). In the following, we thus focus on the challenge of load balancing in PRADA at insert time. Here, we focus on equal distribution of data with DHRs to target nodes as load balancing for indirection information is achieved with the standard mechanisms of key-value based cloud storage systems, e.g., by hashing identifier keys.

In contrast to key-value based cloud storage systems, load balancing in PRADA is more challenging: When processing a create request, the eligible target nodes are not necessarily equal as they might be able to comply with different DHRs. Hence, some eligible nodes might offer rarely supported but often requested requirements. Foreseeing future demands is notoriously difficult [49], thus we suggest to make the load balancing decision based on the current load of the nodes. This requires all nodes to be aware of the load of the other nodes in the cluster. Cloud storage systems typically already exchange this information or can be extended to do so, e.g., using efficient gossiping protocols [50]. We utilize this load information in PRADA as follows. To select the target nodes from the set of eligible nodes, PRADA first checks if any of the responsible nodes are also eligible to become a target node and selects those as target nodes first. This allows us to increase the performance of CRUD requests as we avoid the indirection layer in this case. For the remaining target nodes, PRADA selects those with the lowest load. To have access to more timely load information, each node in PRADA keeps track of all create requests it is involved with. Whenever a node itself stores new data or sends data for storage to other nodes, it increments temporary load information for the respective node. This temporary node information is used to bridge the time between two updates of the load information. As we will show in Section 9.2, this approach enables PRADA to adapt to different usage scenarios and quickly achieve a (nearly) equally balanced storage cluster.

7 FAILURE RECOVERY

When introducing support for DHRs to cloud storage systems, we must ensure not to break their failure recovery mechanisms. With PRADA, we specifically need to take care of dangling references, i.e., a reference pointing to a node that does not store the corresponding data, and unreferenced data, i.e., data stored on a target node without an existing corresponding reference. These inconsistencies could stem from failures during the (modified) CRUD operations as well as from actions that are triggered by DHRs, e.g., deletions forced by DHRs require propagation of meta information to corresponding responsible nodes.

Create. Create requests require to transmit data to the target node and inform the responsible node to store the reference. Failures during these operations can be recognized by the coordinator by missing acknowledgments. Resolving these errors requires a rollback and/or reissuing actions, e.g., selecting a new target node and updating the reference. Still, also the coordinator itself can fail during the process, which may lead to unreachable data. As such failures happen only rarely, we suggest refraining from including corresponding consistency checks directly into create operations [51]. Instead, the client detects failures of the coordinator due to absent acknowledgments. In this case, the client informs all eligible nodes to remove the unreferenced data and reissues the create operation through another coordinator.

Read. In contrast to the other operations, a read request does not change any state in the cloud storage system. Therefore, read requests are simply reissued in case of a failure (identified by a missing acknowledgment) and no further error handling is required.

Update. Although update operations are more complex than create operations, failure handling can happen analogously. As the responsible node updates its reference only upon reception of the acknowledgment from the new target node, the storage state is guaranteed to remain consistent. Hence, the coordinator can reissue the process using the same or a new target node and perform corresponding cleanups if errors occur. Contrary, if the coordinator fails, information on the potentially new target node is lost. Similar to create operations, the client resolves this error by informing all eligible nodes about the failure. Subsequently, the responsible nodes trigger a cleanup to ensure a consistent storage state.

Delete. When deleting data, a responsible node may delete a reference but fail in informing the target node to carry out the delete. Coordinator and client easily detect this error through the absence of the corresponding acknowledgment. Again, the coordinator or client then issue a broadcast message to delete the corresponding data item from the target node. This approach is more reasonable than directly incorporating consistency checks for all delete operations as such failures occur only rarely [51].

Propagating target node actions. CRUD operations are triggered by clients. However, data deletion or relocation, which may result in dangling references or unreferenced data, can also be triggered by the storage cluster or by DHRs that, e.g., specify a maximum lifetime for data. To keep the state of the cloud storage system consistent, storage nodes perform data deletion and relocation through a coordinator as well, i.e., they select one of the other nodes to perform update and delete operations on their behalf. Thus, the correct execution of deletion and relocation tasks can be monitored and repair operations can be triggered. In case either the initiating storage node or the coordinator fails while processing a query, the same mitigations as for CRUD operations (triggered by clients) apply. To protect against rare cases in which both, initiating storage node and coordinator, fail while processing an operation, storage system operators can optionally employ commit logs, e.g., based on Cassandra's atomic batch log [52].

8 IMPLEMENTATION

For the practical evaluation of our approach, we fully implemented PRADA on top of Cassandra [26] (our implementation is available under the Apache License [20]). Cassandra is a distributed database that is actively employed as a key-value based cloud storage system by more than 1500 companies with deployments of up to 75 000 nodes [53] and offers high scalability even over multiple data centers [54], which makes it especially suitable for our scenario. Cassandra also implements advanced features that go beyond simple key-value storage such as column-orientation and queries over ranges of keys, which allows us to showcase the flexibility and adaptability of our design. Data in Cassandra is divided into multiple logical databases, called *key spaces*. A key space consists of tables which are called *column families* and contain rows and columns. Each node knows about all other nodes and their ranges of the hash table. Cassandra uses the gossiping protocol Scuttlebutt [50] to efficiently distribute this knowledge as well as to detect node failure and exchange node state, e.g., load information. Our implementation is based on Cassandra 2.0.5, but our design conceptually also works with newer versions.

Information stores. PRADA relies on three information stores: the global capability store as well as relay and target stores (cf. Section 3). We implement these as individual key spaces in Cassandra as detailed in the following. First, we realize the *global capability store* as a key space that is globally replicated among all nodes (i.e., each node stores a full copy of the capability store to improve performance of create operations) initialized at the same time as the cluster. On this key space, we create a column family for each DHR type (as introduced in Section 2.2). When a node joins the cluster, it inserts all DHR properties it supports for each DHR type (as locally configured by operator of the cloud storage system) into the corresponding column family. This information is then automatically replicated to all other nodes in the cluster by the replication strategy of the corresponding key space. For each regular key space of the database, we additionally create a corresponding *relay store* and *target store* as key spaces. Here, the *relay store* inherits the replication factor and replication strategy from the corresponding regular key space to achieve replication for PRADA as detailed in Section 5, i.e., the relay store will be replicated in exactly the same way as the regular key store. Hence, for each column family in the corresponding key space, we create a column family in the relay key space that acts as the relay store. We follow a similar approach for realizing the *target store*, i.e., we create for each key space a corresponding key space to store actual data. For each column family in the original key space, we create an exact copy in the target key space to act as the target store. However, to ensure that DHRs are adhered to, we implement a DHR-agnostic replication mechanism for the target store and use the relay store to address data.

While the global capability store is created when the cluster is initiated, relay and target stores have to be created whenever a new key space and column family is created, respectively. To this end, we hook into Cassandra's `CreateKeyspaceStatement` class for detecting requests for creating key spaces and column families and subsequently initialize the corresponding relay and target stores.

Creating data and load balancing. To allow clients to specify their DHRs when inserting or updating data, we support the specification of arbitrary DHRs in textual form for `INSERT` requests (cf. Section 2.1). To this end, we add an optional postfix `WITH REQUIREMENTS` to `INSERT` statements by extending the grammar from which parser and lexer for CQL3 [55], the SQL-like query language of Cassandra, are generated using ANTLR [56]. Using the `WITH REQUIREMENTS` statement, arbitrary DHRs can be specified separated by the keyword `AND`, e.g., `INSERT ... WITH REQUIREMENTS location = { 'DE', 'FR', 'UK' } AND encryption = { 'AES-256' }`. In this example, any node located in Germany, France, or the United Kingdom that supports AES-256 encryption is eligible to store the inserted data. This approach enables users to specify any DHRs covered by our formalized model of DHRs (cf. Section 2.2).

To detect and process DHRs in create requests (cf. Section 4), we extend Cassandra's `QueryProcessor`, specifically its `getStatement` method for processing `INSERT` requests. When processing requests with DHRs (specified using the `WITH REQUIREMENTS` statement), we base our selection of eligible nodes on the global capability store. Nodes are eligible to store data with a given set of DHRs if they provide at least one of the specified properties for each requested type (e.g., one out of multiple permitted locations). We prioritize nodes that Cassandra would pick without DHRs, as this speeds up reads for the corresponding key later on, and otherwise choose nodes according to our load balancer (cf. Section 6). Our load balancing implementation relies on Cassandra's gossiping mechanism [26], which maintains a map of all nodes together with their corresponding loads. We access this information using Cassandra's `getLoadInfo` method and extend the load information with local estimators for load changes. Whenever a node sends a create request or stores data itself, we update the corresponding local estimator with the size of the inserted data. To this end, we hook into the methods that are called when data is modified locally or forwarded to other nodes, i.e., the corresponding methods in Cassandra's `ModificationStatement`, `RowMutationVerbHandler`, and `StorageProxy` classes as well as our methods for processing requests with DHRs.

Reading data. To allow reading redirected data as described in Section 4, we modify Cassandra's `ReadVerbHandler` class for processing read requests at the responsible node. This handler is called whenever a node receives a read request from the coordinator and allows us to check whether the current node holds a reference to another target node for the requested key by locally checking the corresponding column family within the relay store. If no reference exists, the node continues with a standard read operation. Otherwise, the node forwards a modified read request to one deterministically selected target node (cf. Section 5) using Cassandra's `sendOneWay` method, in which it requests the data from the respective target on behalf of the coordinator. Subsequently, the target nodes send the data directly to the coordinator node (whose identifier is included in the request). To correctly resolve references to data for which the coordinator of a query is also the responsible node, we additionally add corresponding checks to the `LocalReadRunnable` subclass of `StorageProxy`.

9 EVALUATION

We perform benchmarks to quantify query completion times, storage space, and consumed traffic. Furthermore, we study PRADA's load behavior through simulation and show PRADA's applicability in three real-world use cases.

9.1 Benchmarks

First, we benchmark query completion time, consumed storage space, and bandwidth consumption. In all settings, we compare the performance of PRADA with the performance of an unmodified Cassandra installation as well as PRADA*, a system running PRADA but receiving only data without DHRs. This enables us to verify that data without DHRs is indeed not impaired by PRADA.

We set up a cluster of 10 nodes interconnected via a gigabit Ethernet switch. All nodes are equipped with an Intel Core 2 Q9400 and 4 GB RAM as well as either 160 GB or 500 GB storage and run either Ubuntu 14.04 or 16.04. We assign each node a distinct artificial DHR property to avoid potential bias resulting from using only one specific DHR type (such as storage location). When inserting or updating data, clients request a set of exactly three of the available properties uniformly randomly. Each row of data consists of 200 B of uniformly random data (+ 20 B for the key), spread over 10 columns. These are rather conservative numbers as the relative overhead of PRADA decreases with increasing storage size. For each result, we performed 5 runs, each with 1000 operations which were performed in one burst, i.e., as quickly as could be handled by the coordinator. In the following, we depict the mean value for performing one operation with 99% confidence intervals. We provide further instructions on how to perform these measurements as part of the release of our implementation [20].

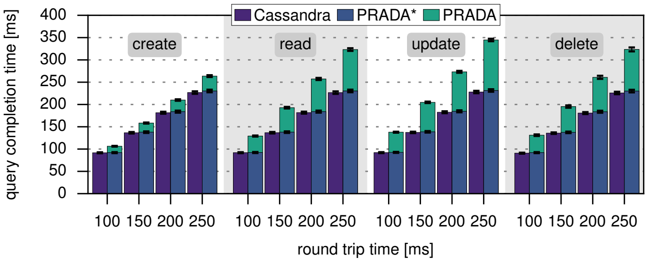


Fig. 5. **Query time vs. RTT.** PRADA constitutes limited overhead for operations on data with DHRs, while data without DHRs is not impacted.

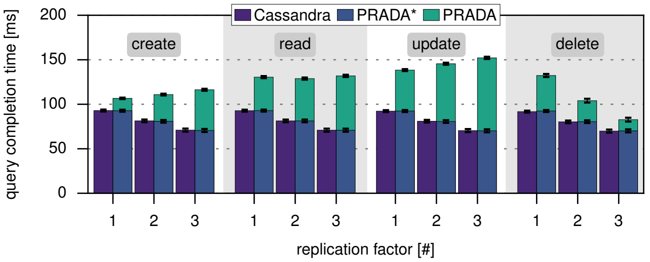


Fig. 6. **Query time vs. replication.** Create and update in PRADA show modest overhead for increasing replicas due to larger message sizes.

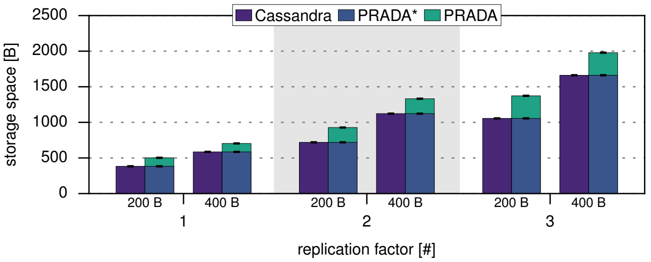


Fig. 7. **Storage vs. replication.** PRADA constitutes only constant overhead per DHR affected replica, while not affecting data without DHRs.

Query completion time. The query completion time (QCT) denotes the time the coordinator takes for processing a query, i.e., from receiving it until sending the result back to the client. It is influenced by the round-trip time (RTT) between nodes in the cluster and the replication factor.

We first study the influence of RTTs on QCT for a replication factor $r = 1$. To this end, we artificially add latency to outgoing packets for inter-cluster communication using `netem` [57] to emulate RTTs of 100 to 250 ms. Our choice covers RTTs observed in communication between cloud data centers around the world [58] and verified through measurements in the Microsoft Azure cloud. In Figure 5, we depict the QCTs for the different CRUD operations and RTTs. We make two observations. First, QCTs of PRADA* are indistinguishable from those of Cassandra, which implies that data without DHRs is not impaired by PRADA. Second, the additional overhead of PRADA lies between 15.4 to 16.2% for create, 40.5 to 42.1% for read, 48.9 to 50.5% for update, and 44.3 to 44.8% for delete. The overheads for read, update, and delete correspond to the additional 0.5 RTT introduced by the indirection layer and is slightly worse for updates as data stored at potentially old target nodes needs to be deleted. QCTs below the RTT result from corner cases where the coordinator is also responsible for storing data.

From now on, we fix RTTs at 100 ms and study the impact of replication factors $r = 1, 2,$ and 3 on QCTs as shown in Figure 6. Again, we observe that the QCTs of PRADA* and Cassandra are indistinguishable. For increasing replication factors, the QCTs for PRADA* and Cassandra reduce as it becomes more likely that the coordinator also stores the data. In this case, Cassandra optimizes queries. When considering the overhead of PRADA, we witness that the QCTs for creates (overhead increasing from 14 to 46 ms), reads (overhead increasing from 38 to 61 ms), and updates (overhead increasing from 46 to 80 ms) cannot benefit from these optimizations, as this would require the coordinator to be responsible and target node at the same time, which happens only rarely. Furthermore, the increase in QCTs for creates and updates results from the overhead of handling r references at r nodes, while the increase for reads corresponds to the additional 0.5 RTT for the indirection layer. For deletes, the overhead decreases from 41 to 12 ms for an increasing replication factor, which results from an increased likelihood that the coordinator node is at least either responsible or target node, avoiding additional communication.

Consumed storage space. To quantify the additional storage space required by PRADA, we measure the consumed storage space after data has been inserted, using the `cfstats` option of Cassandra's `nodetool` utility. To this end, we conduct insertions for payload sizes of 200 and 400 B (plus 20 B for the key), i.e., we fill 10 columns with 20 respective 40 B payload in each query, with replication factors of $r = 1, 2,$ and 3. In real-world use cases, we observe, e.g., a mean payload size of 312 B for an IoT data platform (cf. Section 9.3). We divide the total consumed storage space per run by the number of insertions and show the mean consumed storage space per inserted row over all runs in Figure 7. Cassandra requires 383 B to store 200 B of payload and 585 B for a payload of 400 B. Each additional replica increases the required storage space by roughly 90%. PRADA adds a constant overhead of roughly 110 B per replica. While the precise overhead of PRADA depends on the encoding of relay information, the important observation here is that it does not depend on the size of the stored data. Even for extremely small payload sizes, e.g., a mean payload size of 133 B in a microblogging use case (cf. Section 9.3), PRADA adds only an additional relative storage overhead of roughly 38% on top of an overhead of more than 136% already added by Cassandra. When considering larger payload sizes, the storage overhead of PRADA becomes negligible, e.g., when storing emails with a mean size of 3626 B (cf. Section 9.3) where the overhead for indirection information amounts to only 3% of the data size.

Bandwidth consumption. We measure the traffic consumed by the individual CRUD operations by hooking into the `writeConnected` method to be able to filter out background traffic such as gossiping. Figure 8 depicts the mean total generated message payload per single operation averaged over 5 runs with 2000 operations each for an RTT of 100 ms. Our results show that using PRADA comes at the cost of an overhead that scales with the replication factor. When considering Cassandra and PRADA*, we observe that the consumed traffic for read operations does only slightly increase when raising the replication factor. This results from an optimization in Cassandra that requests the data only from one replica and probabilistically compares only digests of the data held by the other replicas to perform post-request consistency checks. Furthermore, with increasing replication factors, it becomes more likely that the coordinator also stores the data and thus no communication is necessary, while PRADA requires the coordinator to be responsible and target node at the same time, which happens only rarely. For the other operations, the overhead introduced by our indirection layer ranges from 0.7 to 0.9 kB for a replication factor of 3. For a replication factor of 1, the highest overhead introduced by PRADA peaks at 0.2 kB. Thus, we conclude that the traffic overhead of PRADA allows for a practical operation in cloud storage systems.

9.2 Load Distribution

To quantify the impact of PRADA on the load distribution of the overall cloud storage system, we rely on a simulation approach as this enables a thorough analysis of the load distribution and considering a wide range of scenarios.

Simulation setup. As we are solely interested in the load behavior, we implemented a custom simulator in Python, which models the characteristics of Cassandra with respect to network topology, data placement, and gossip behavior. Using the simulator, we realize a cluster of n nodes, which are equally distributed among the key space [52] and insert m data items with uniformly random keys. For simplicity, we assume that all data items are of the same size. The nodes operate Cassandra's gossip protocol [50], i.e., synchronize with one random node every second and update its own load information every 60 s. We randomize the initial offset before the first gossip message for each node individually, as in reality not all nodes perform the gossip at the same point in time. We repeat each measurement 10 times with different random seeds [59] and show the mean of the load balance $\mathfrak{L}$ (cf. Section 6) with 99% confidence intervals.

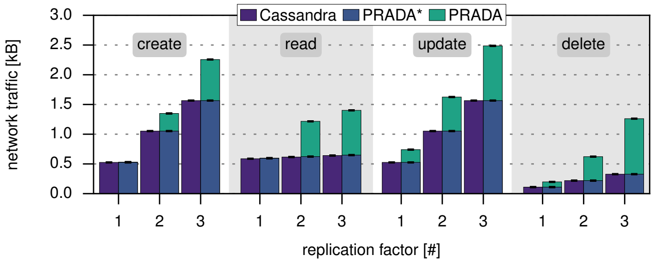


Fig. 8. **Traffic vs. replication.** Data without DHRs is not affected by PRADA. Replicas increase the traffic overhead introduced by DHRs.

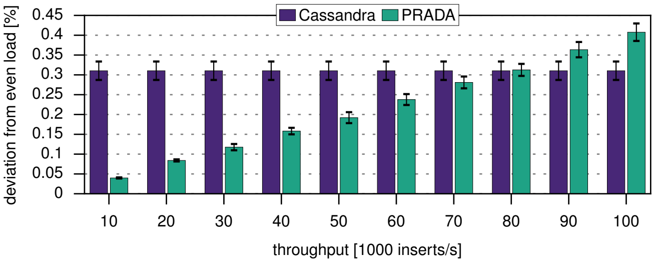


Fig. 9. **Load balance vs. throughput.** Load balance in PRADA depends on throughput of inserts. Even for high throughput it stays below 0.5%.

Influence of throughput. We expect the load distribution to be influenced by the freshness of the load information as gossiped by other nodes, which correlates with the throughput of create requests. A lower throughput results in less data inserted between two load information updates and hence the load information remains relatively fresher. To study this effect, we simulate different insertion throughputs to vary the gossiping delay. We simulate a cluster with 100 nodes and 10^7 create requests, each accompanied by a DHR. Even for high throughput, this produces enough data to guarantee at least one gossip round. To challenge the load balancer, we synthetically create two types of DHRs with two properties, each supported by half of the nodes such that each combination of the properties of the two types of DHRs is supported by 25 nodes. For each create request we randomly select one of the resulting possible DHRs, i.e., demanding one property for one or two of the DHRs types.

Figure 9 shows the deviation from an even load for increasing throughput compared with a traditional Cassandra cluster. Additionally, we calculate the optimal solution under a posteriori knowledge by formulating the corresponding quadratic program for minimizing the load balance $\mathfrak{L}$ and solving it using CPLEX [60]. In all cases, we observe that the resulting optimal load balance is 0, i.e., all nodes are loaded exactly equal, and hence omit these results in the plot. Seemingly large confidence intervals result from the high resolution of our plot (PRADA deviates less than 0.5% from even load). The results show that PRADA even outperforms Cassandra for smaller throughputs (load imbalance of Cassandra results from hashing) and the introduced load imbalance stays well below 0.5% in all scenarios, even for a high throughput of 100 000 insertions/s (Dropbox processed less than 20 000 insertions/s on average in June 2015 [61]). These results indicate that frequent updates of node state improve load balance for PRADA.

Influence of DHR fit. In PRADA, one of the core influence factors on the load distribution is the accordance of clients' DHRs with the properties provided by storage nodes. If the distribution of DHRs heavily deviates from the distribution of DHRs supported by the storage nodes, it is impossible to achieve an even load distribution. To study this aspect, we consider a scenario where each node has a storage location and clients request exactly one of the available storage locations. We simulate a cluster of 1000 nodes that are geographically distributed according to the IP address ranges of Amazon Web Services [62] (North America: 64%, Europe: 17%, Asia-Pacific: 16%, South America: 2%, China: 1%). First, we insert data with DHRs whose distribution exactly matches the distribution of nodes. Subsequently, we worsen the accuracy of fit by subtracting 10 to 100% from the location with the most nodes (i.e., North America) and proportionally distribute this demand to the other locations (in the extreme setting, North America: 0%, Europe: 47.61%, Asia-Pacific: 44.73%, South America: 5.74%, and China: 1.91%). We simulate 10^7 insertions at a throughput of 20 000 insertions/s. For comparison, we calculate the optimal load using a posteriori knowledge by equally distributing the data on the nodes of each location. Our results are depicted in Figure 10. We derive two insights from this experiment: i) the deviation from an even cluster load scales linearly with decreasing accordance of clients' DHRs with node capabilities and ii) in all considered settings PRADA manages to achieve a cluster load that is close to the theoretical optimum. Hence, PRADA's approach of load balancing can indeed adapt to the challenges imposed by complying with DHRs in cloud storage systems.

9.3 Applicability

We show the applicability of PRADA by realizing three real-world use cases: a microblogging system, a distributed email management system, and an IoT platform. To create a realistic evaluation environment, we use a globally distributed cloud storage consisting of 10 nodes on top of the Microsoft Azure cloud platform [63]. More specifically, we utilize virtual machine instances of type D2s v3, each equipped with 2 virtual CPUs, 8 GB RAM, 30 GB storage, and Ubuntu 16.04 as operating system. The virtual machines are globally distributed among 10 distinct regions: asia-east, asia-southeast, canada-central, europe-north, europe-west, japan-east, us-central, us-east, us-southcentral, and us-west2. In case of read timeouts, e.g., due to temporary connection problems, we resubmit the corresponding query. The release of our implementation contains further information on how to perform these measurements [20].

Microblogging. Microblogging services such as Twitter frequently utilize cloud storage systems to store messages. To evaluate the impact of PRADA on such services, we use the database layout of Twissandra [64], an exemplary implementation of a microblogging service for Cassandra, and real tweets from the twitter7 dataset [65]. For each user, we uniformly at random select one of the storage locations and attach it as DHR to all their tweets. We perform our measurements using a replication factor of $r = 1$ and measure the QCTs for randomly chosen users for retrieving their userline (most recent messages of a user) and their timeline (most recent messages of all users a user follows). To this end, we insert 2 million tweets from the twitter7 dataset [65] and randomly select 1000 users among those users who have at least 50 tweets in our dataset. For the userline measurement, we request 50 consecutive tweets of each selected user. As the twitter7 dataset does not contain follower relationships between users, we request 50 random tweets for the timeline measurements of each selected user.

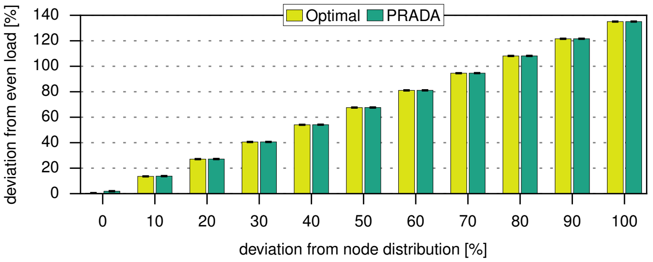


Fig. 10. **Load balance vs. node distribution.** PRADA's load balance shows optimal behavior, but depends on node distribution.

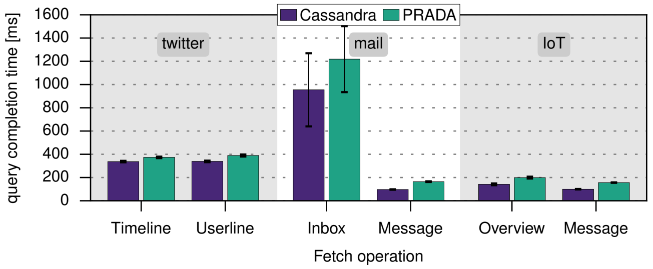


Fig. 11. **Usecase evaluation.** Adding DHRs to tweets delays query completion by 11 to 15%. Also for email storage and IoT data, accounting for compliance with DHRs results in acceptable overheads.

Our results in Figure 11 (left) show that the runtime overhead of supporting DHRs for microblogging in a globally distributed cluster corresponds to a 11% (15%) increase in query completion time for fetching the timeline (userline). Here, PRADA especially benefits from the fact that identifiers are spread along the cluster and thus the unmodified Cassandra also has to contact a large number of nodes. Our results show that PRADA can be applied to offer support for DHRs in microblogging at reasonable costs with respect to query completion time. Especially when considering that not each tweet will likely be accompanied by DHRs, this modest overhead is well worth the additional functionality.

Email storage. Email providers increasingly move storage of emails to the cloud [66]. To study the impact of supporting DHRs on emails, we analyzed Cassandra-backed email systems such as Apache James [67] and ElasticInbox [68] and derived a common database layout consisting of one table for meta data (overview of a complete mailbox) and one table for full emails. To create a realistic scenario, we utilize the Enron email dataset [69], consisting of about half a million emails of 150 users. For each user, we uniformly at random select one of the available storage locations as DHR for their emails and meta information.

Figure 11 (middle) compares the mean QCTs per operation of Cassandra and PRADA for fetching the *overview of the mailbox* for all 150 users and fetching 10 000 randomly selected *individual emails*. For fetching of mailboxes, we observe overlapping, rather large confidence intervals resulting from the small number of operations (only 150 mailboxes) and huge differences in mailbox sizes , ranging from 35 to 28 465 messages. While we cannot derive a definitive statement (at the 99% confidence level) from these results, the mean QCTs for fetching the overview of a mailbox seem to suggest a notable yet acceptable overhead for using PRADA. When considering the fetching of individual messages, we observe an overhead of 70% for PRADA's indirection step, increasing QCTs from 97 to 164 ms. Hence, we can provide compliance with DHRs for email storage with a reasonable increase of 67 ms for fetching individual emails and a likely increase in the time required for generating an overview of all emails in the mailbox in the order of 28%.

IoT platform. The Internet of Things (IoT) leads to a massive growth of collected data which is often stored in the cloud [70], [71]. Literature proposes to attach per-data item DHRs to IoT data to preserve privacy [31], [71], [72]. To study the applicability of PRADA in this setting, we collected frequency and size of authentic IoT data from the IoT data sharing platform dweet.io [73]. Our data set contains 1.84 million IoT messages of size 72 B to 9.73 KB from 2889 devices. To protect the privacy of people monitored by these devices, we replaced all payload information with random data. For each device, we uniformly at random assign one of the storage locations as DHR for the collected data.

In Figure 11 (right), we depict the mean QCTs per operation of Cassandra and PRADA for retrieving the *overview of all IoT data* for each of the 2889 devices as well as for accessing 10 000 randomly selected *single IoT messages*. The varying amount of sensor data that different IoT devices offer leads to a slightly varying QCT for fetching of IoT device data overviews, similar to mailbox fetching (see above). The overhead for adhering to DHRs with PRADA in the IoT use case totals to 41% for the fetching of a device's IoT data overview and 57% for a single IoT message, corresponding to the 0.5 RTT added by the indirection layer. We consider these overheads still appropriate given the inherent private nature of most IoT data and the accompanying privacy risks which can be mitigated with DHRs.

10 RELATED WORK

We categorize our discussion of related work by the different types of DHRs they address. In addition, we discuss approaches for providing assurance that DHRs are respected.

Distributing storage of data. To enforce storage location requirements, a class of related work proposes to split data between different storage systems. Wüchner et al. [12] and CloudFilter [18] add proxies between clients and operators to transparently distribute data to different cloud storage providers according to DHRs, while NubiSave [19] allows combining resources of different storage providers to fulfill individual redundancy or security requirements of clients. These approaches can treat individual storage systems only as black boxes. Consequently, they do not support fine-grained DHRs within the database system itself and are limited to a small subset of DHRs.

Sticky policies. Similar to our idea of specifying DHRs, the concept of sticky policies proposes to attach usage and obligation policies to data when it is outsourced to third-parties [31]. In contrast to our work, sticky policies mainly concern the purpose of data usage, which is primarily realized using access control. One interesting aspect of sticky policies is their ability to make them "stick" to the corresponding data using cryptographic measures which could also be applied to PRADA. In the context of cloud computing, sticky policies have been proposed to express requirements on the security and geographical location of storage nodes [32]. However, so far it has been unclear how this could be realized efficiently in a large and distributed storage system. With PRADA, we present a mechanism to achieve this goal.

Policy enforcement. To enforce privacy policies when accessing data in the cloud, Betgé-Brezetz et al. [13] monitor access of virtual machines to shared file systems and only allow access if a virtual machine is policy compliant. In contrast, Itani et al. [14] propose to leverage cryptographic coprocessors to realize trusted and isolated execution environments and enforce the encryption of data. Espling et al. [15] aim at allowing service owners to influence the placement of their virtual machines in the cloud to realize specific geographical deployments or provide redundancy through avoiding co-location of critical components. These approaches are orthogonal to our work, as they primarily focus on enforcing policies when processing data, while PRADA addresses the challenge of supporting DHRs when storing data in cloud storage systems.

Location-based storage. Focusing exclusively on location requirements, Peterson et al. [16] introduce the concept of data sovereignty with the goal to provide a guarantee that a provider stores data at claimed physical locations, e.g., based on measurements of network delay. Similarly, LoSt [17] enables verification of storage locations based on a challenge-response protocol. In contrast, PRADA focuses on the more fundamental challenge of realizing the functionality for supporting arbitrary DHRs.

Controlling placement of data. Primarily focusing on distributed hash tables, SkipNet [74] enables control over data placement by organizing data mainly based on string names. Similarly, Zhou et al. [75] utilize location-based node identifiers to encode physical topology and hence provide control over data placement at a coarse grain. In contrast to PRADA, these approaches need to modify the identifier of data based on the DHRs, i.e., knowledge about the specific DHRs of data is required to locate it. Targeting distributed object-based storage systems, CRUSH [76] relies on hierarchies and data distribution policies to control placement of data in a cluster. These data distribution policies are bound to a predefined hierarchy and hence cannot offer the same flexibility as PRADA. Similarly, Tenant-Defined Storage [77] enables clients to store their data according to DHRs. However and in contrast to PRADA, all data of one client needs to have the same DHRs. Finally, SwiftAnalytics [78] proposes to control the placement of data to speed up big data analytics. Here, data can only be put directly on specified nodes without the abstraction provided by PRADA's approach of supporting DHRs.

Hippocratic databases. Hippocratic databases store data together with a purpose specification [79]. This allows them to enforce the purposeful use of data using access control and to realize data retention after a certain period. Using Hippocratic databases, it is possible to create an auditing framework to check if a database is complying with its data disclosure policies [33]. However, this concept only considers a single database and not a distributed setting where storage nodes have different data handling capabilities.

Assurance. To provide assurance that storage operators adhere to DHRs, de Oliveira et al. [80] propose an architecture to automate the monitoring of compliance to DHRs when transferring data. Bacon et al. [34] and Pasquier et al. [5] show that this can also be achieved using information flow control. Similarly, Massonet et al. [41] propose a monitoring and audit logging architecture in which the infrastructure provider and service provider collaborate to ensure data location compliance. These approaches are orthogonal to our work and could be used to verify that operators of cloud storage systems run PRADA in an honest way and error-free.

11 Discussion and Conclusion

Accounting for compliance with data handling requirements (DHRs), i.e., offering control over where and how data is stored in the cloud, becomes increasingly important due to legislative, organizational, or customer demands. Despite these incentives, practical solutions to address this need in existing cloud storage systems are scarce. In this paper, we proposed PRADA, which allows clients to specify a comprehensive set of fine-grained DHRs and enables cloud storage operators to enforce them. Our results show that we can indeed achieve support for DHRs in cloud storage systems. Of course, the additional protection and flexibility offered by DHRs comes at a price: We observe a moderate increase for query completion times, while achieving constant storage overhead and upholding a near optimal storage load balance even in challenging scenarios.

Importantly, however, data without DHRs is not impaired by PRADA. When a responsible node receives a request for data without DHRs, it can *locally* check that no DHRs apply to this data: For create requests, the `INSERT` statement either contains DHRs or not, which can be checked efficiently and locally. In contrast, for read, update, and delete requests, PRADA performs a simple and local check whether a reference to a target node for this data exists. The overhead for this step is comparable to executing an `if` statement and hence negligible. Only if a reference exists, which implies that the data was inserted with DHRs, PRADA induces overhead. Our extensive evaluation confirms that, for data without DHRs, PRADA shows the same query completion times, storage overhead, and bandwidth consumption as an unmodified Cassandra system in all considered settings (indistinguishable results for Cassandra and PRADA* in Figures 5 to 8.) Consequently, clients can choose (even at a granularity of individual data items), if DHRs are worth a modest performance decrease.

PRADA's design is built upon a transparent indirection layer, which effectively handles compliance with DHRs. This design decision limits our solution in three ways. First, the overall achievable load balance depends on how well the nodes' capabilities to fulfill certain DHRs matches the actual DHRs requested by the clients. However, for a given scenario, PRADA is able to achieve nearly optimal load balance as shown in Figure 10. Second, indirection introduces an overhead of 0.5 round-trip times for reads, updates, and deletes. Further reducing this overhead is only possible by encoding some DHRs in the key used for accessing data [23], but this requires everyone accessing the data to be in possession of the DHRs, which is unlikely. A fundamental improvement could be achieved by replicating all relay information to all nodes of the cluster, but this is viable only for small cloud storage systems and does not offer scalability. We argue that indirection can likely not be avoided, but still pose this as an open research question. Third, the question arises how clients can be assured that an operator indeed enforces their DHRs and no errors in the specification of DHRs have occurred. This has been widely studied [16], [33], [41], [80] and the proposed approaches such as audit logging, information flow control, and provable data possession can also be applied to PRADA.

While we limit our approach for providing data compliance in cloud storage to key-value based storage systems, the key-value paradigm is also general enough to provide a practical starting point for storage systems that are based on different paradigms. Additionally, the design of PRADA is flexible enough to extend (with some more work) to other storage systems. For example, Google's globally distributed database Spanner (rather a multi-version database than a key-value store) allows applications to influence data locality (to increase performance) by carefully choosing keys [28]. PRADA could be applied to Spanner by modifying Spanner's approach of directory-bucketed key-value mappings. Likewise, PRADA could realize data compliance for distributed main memory databases, e.g., VoltDB, where tables of data are partitioned horizontally into shards [29]. Here, the decision on how to distribute shards over the nodes in the cluster could be taken with DHRs in mind. Similar adaptations could be performed for commercial products, such as Clustrix [30], that separate data into slices.

To conclude, PRADA resolves a situation, i.e., missing support for DHRs, that is disadvantageous to both clients *and* operators of cloud storage systems. By offering the enforcement of arbitrary DHRs when storing data in cloud storage systems, PRADA enables the use of cloud storage systems for a wide range of clients who previously had to refrain from outsourcing storage, e.g., due to compliance with applicable data protection legislation. At the same time, we empower cloud storage operators with a practical and efficient solution to handle differences in regulations and offer their services to new clients.

Acknowledgments

The authors would like to thank Annika Seufert for support with the simulations. This work has received funding from the European Union's Horizon 2020 research and innovation program 2014-–2018 under grant agreement No. 644866 (SSICLOPS) and from the Excellence Initiative of the German federal and state governments. This article reflects only the authors' views and the funding agencies are not responsible for any use that may be made of the information it contains.

References

[1] R. Gellman, "Privacy in the Clouds: Risks to Privacy and Confidentiality from Cloud Computing," World Privacy Forum, 2009.
[2] S. Pearson and A. Benameur, "Privacy, Security and Trust Issues Arising from Cloud Computing," in *IEEE CloudCom*, 2010.

[3] United States Congress, “Gramm-Leach-Bliley Act (GLBA),” Pub.L. 106-102, 113 Stat. 1338, 1999.

[4] D. Song *et al.*, “Cloud Data Protection for the Masses,” *Computer*, vol. 45, no. 1, 2012.

[5] T. F. J. M. Pasquier *et al.*, “Information Flow Audit for PaaS Clouds,” in *IEEE IC2E*, 2016.

[6] V. Abramova and J. Bernardino, “NoSQL Databases: MongoDB vs Cassandra,” in *C3S2E*, 2013.

[7] R. Buyya, R. Ranjan, and R. N. Calheiros, “InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services,” in *ICA3PP*, 2010.

[8] D. Bernstein *et al.*, “Blueprint for the Intercloud - Protocols and Formats for Cloud Computing Interoperability,” in *ICIW*, 2009.

[9] Intel IT Center, “Peer Research: What’s Holding Back the Cloud?” Tech. Rep., 2012.

[10] D. Catteddu and G. Hogben, “Cloud Computing – Benefits, Risks and Recommendations for Information Security,” European Network and Information Security Agency (ENISA), 2009.

[11] M. Henze, R. Hummen, and K. Wehrle, “The Cloud Needs Cross-Layer Data Handling Annotations,” in *IEEE S&P Workshops*, 2013.

[12] T. Wüchner, S. Müller, and R. Fischer, “Compliance-Preserving Cloud Storage Federation Based on Data-Driven Usage Control,” in *IEEE CloudCom*, 2013.

[13] S. Betgé-Brezetz *et al.*, “End-to-End Privacy Policy Enforcement in Cloud Infrastructure,” in *IEEE CloudNet*, 2013.

[14] W. Itani, A. Kayssi, and A. Chehab, “Privacy as a Service: Privacy-Aware Data Storage and Processing in Cloud Computing Architectures,” in *IEEE DASC*, 2009.

[15] D. Espling *et al.*, “Modeling and Placement of Cloud Services with Internal Structure,” *IEEE Transactions on Cloud Computing*, vol. 4, no. 4, 2014.

[16] Z. N. J. Peterson, M. Gondree, and R. Beverly, “A Position Paper on Data Sovereignty: The Importance of Geolocating Data in the Cloud,” in *USENIX HotCloud*, 2011.

[17] G. J. Watson *et al.*, “LoSt: Location Based Storage,” in *ACM CCSW*, 2012.

[18] I. Papagiannis and P. Pietzuch, “CloudFilter: Practical Control of Sensitive Data Propagation to the Cloud,” in *ACM CCSW*, 2012.

[19] J. Spillner, J. Müller, and A. Schill, “Creating optimal cloud storage systems,” *Future Generation Computer Systems*, vol. 29, no. 4, 2013.

[20] RWTH Aachen University, “PRADA Source Code Repository,” https://github.com/COMSYS/prada.

[21] M. Henze *et al.*, “Practical Data Compliance for Cloud Storage,” in *IEEE IC2E*, 2017.

[22] P. Samarati and S. De Capitani di Vimercati, “Data Protection in Outsourcing Scenarios: Issues and Directions,” in *ACM ASIACSS*, 2010.

[23] M. Henze *et al.*, “Towards Data Handling Requirements-aware Cloud Computing,” in *IEEE CloudCom*, 2013.

[24] A. Greenberg *et al.*, “The Cost of a Cloud: Research Problems in Data Center Networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, 2008.

[25] G. DeCandia *et al.*, “Dynamo: Amazon’s Highly Available Key-value Store,” in *ACM SOSP*, 2007.

[26] A. Lakshman and P. Malik, “Cassandra: A Decentralized Structured Storage System,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, 2010.

[27] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 3rd ed. Springer, 2011.

[28] J. C. Corbett *et al.*, “Spanner: Google’s Globally-distributed Database,” in *USENIX OSDI*, 2012.

[29] M. Stonebraker and A. Weisberg, “The VoltDB Main Memory DBMS,” *IEEE Data Eng. Bull.*, vol. 36, no. 2, 2013.

[30] Clustrix, Inc., “Scale-Out NewSQL Database in the Cloud,” http://www.clustrix.com/.

[31] S. Pearson and M. C. Mont, “Sticky Policies: An Approach for Managing Privacy across Multiple Parties,” *Computer*, vol. 44, no. 9, 2011.

[32] S. Pearson, Y. Shen, and M. Mowbray, “A Privacy Manager for Cloud Computing,” in *CloudCom*, 2009.

[33] R. Agrawal *et al.*, “Auditing Compliance with a Hippocratic Database,” in *VLDB*, 2004.

[34] J. Bacon *et al.*, “Information Flow Control for Secure Cloud Computing,” *IEEE Transactions on Network and Service Management*, vol. 11, no. 1, 2014.

[35] U. Rührmair *et al.*, “Virtual Proofs of Reality and their Physical Implementation,” in *IEEE S&P*, 2015.

[36] United States Congress, “Health Insurance Portability and Accountability Act of 1996 (HIPAA),” Pub.L. 104–191, 110 Stat. 1936, 1996.

[37] “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation),” L119, 4/5/2016, 2016.

[38] PCI Security Standards Council, “Payment Card Industry (PCI) Data Security Standard – Requirements and Security Assessment Procedures, Version 3.1,” 2015.

[39] R. Buyya *et al.*, “Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility,” *Future Generation Computer Systems*, vol. 25, no. 6, 2009.

[40] T. Ristenpart *et al.*, “Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds,” in *ACM CCS*, 2009.

[41] P. Massonet *et al.*, “A Monitoring and Audit Logging Architecture for Data Location Compliance in Federated Cloud Infrastructures,” in *IEEE IPDPS Workshops*, 2011.

[42] United States Congress, “Sarbanes-Oxley Act (SOX),” Pub.L. 107–204, 116 Stat. 745, 2002.

[43] A. Mantelero, “The EU Proposal for a General Data Protection Regulation and the roots of the ‘right to be forgotten’,” *Computer Law & Security Review*, vol. 29, no. 3, 2013.

[44] H. A. Jäger *et al.*, “Sealed Cloud – A Novel Approach to Safeguard against Insider Attacks,” in *Trusted Cloud Computing*. Springer, 2014.

[45] J. Singh *et al.*, “Regional clouds: technical considerations,” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-863, 2014.

[46] S. Pearson, “Taking Account of Privacy when Designing Cloud Computing Services,” in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*. IEEE, 2009.

[47] E. Barker, “Recommendation for Key Management – Part 1: General (Revision 4),” NIST Special Publication 800-57, National Institute of Standards and Technology, 2015.

[48] A. Corradi, L. Leonardi, and F. Zambonelli, “Diffusive Load-Balancing Policies for Dynamic Applications,” *IEEE Concurrency*, vol. 7, no. 1, 1999.

[49] L. Rainie and J. Anderson, “The Future of Privacy,” Pew Research Center, http://www.pewinternet.org/2014/12/18/future-of-privacy/, 2014.

[50] R. van Renesse *et al.*, “Efficient Reconciliation and Flow Control for Anti-entropy Protocols,” in *LADIS*, 2008.

[51] J. K. Nidzwetzki and R. H. Güting, “Distributed SECONDO: A Highly Available and Scalable System for Spatial Data Processing,” in *SSTD*, 2015.

[52] DataStax, Inc., “Apache Cassandra^{TM} 2.0 Documentation,” http://docs.datastax.com/en/cassandra/2.0/pdf/cassandra20.pdf, 2016, last updated: 21 January 2016.

[53] The Apache Software Foundation, “Apache Cassandra,” https://cassandra.apache.org/.

[54] T. Rabl *et al.*, “Solving Big Data Challenges for Enterprise Application Performance Management,” *Proc. VLDB Endow.*, vol. 5, no. 12, 2012.

[55] The Apache Software Foundation, “Cassandra Query Language (CQL) v3.3.1,” https://cassandra.apache.org/doc/cql3/CQL.html, 2015.

[56] T. J. Parr and R. W. Quong, “ANTLR: A predicated-LL(k) parser generator,” *Software: Practice and Experience*, vol. 25, no. 7, 1995.

[57] S. Hemminger, “Network Emulation with NetEm,” in *linux.conf.au*, 2005.

[58] S. Sanghrajka, N. Mahajan, and R. Sion, “Cloud Performance Benchmark Series: Network Performance – Amazon EC2,” Cloud Commons Online, 2011.

[59] J. Walker, “HotBits: Genuine Random Numbers,” http://www.fourmilab.ch/hotbits.

[60] IBM Corporation, “IBM ILOG CPLEX Optimization Studio,” http://www.ibm.com/software/products/en/ibmilogcpleoptistud/.

[61] Dropbox Inc., “400 million strong,” https://blogs.dropbox.com/dropbox/2015/06/400-million-users/, 2015.

[62] Amazon Web Services, Inc., “Amazon Web Services General Reference Version 1.0,” http://docs.aws.amazon.com/general/latest/gr/aws-general.pdf.

[63] Microsoft Corporation, “Microsoft Azure Cloud Computing Platform & Services,” https://azure.microsoft.com/.

[64] "Twissandra," http://twissandra.com/.

[65] J. Yang and J. Leskovec, "Patterns of Temporal Variation in Online Media," in *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining (WSDM)*. ACM, 2011.

[66] K. Giannakouris and M. Smihily, "Cloud computing – statistics on the use by enterprises," Eurostat Statistics Explained, 2014.

[67] The Apache Software Foundation, "Apache James Project," http://james.apache.org/.

[68] "ElasticInbox – Scalable Email Store for the Cloud," http://www.elasticinbox.com/.

[69] B. Klimt and Y. Yang, "Introducing the Enron Corpus," in *First Conference on Email and Anti-Spam (CEAS)*, 2004.

[70] M. Henze *et al.*, "A Comprehensive Approach to Privacy in the Cloud-based Internet of Things," *FGCS*, 2016.

[71] M. Henze *et al.*, "CPPL: Compact Privacy Policy Language," in *ACM WPES*, 2016.

[72] T. Pasquier *et al.*, "Data-centric access control for cloud computing," in *ACM SACMAT*, 2016.

[73] Bug Labs, Inc., "dweet.io – Share your thing like it ain't no thang." https://dweet.io/.

[74] N. J. A. Harvey *et al.*, "SkipNet: A Scalable Overlay Network with Practical Locality Properties," in *USENIX USITS*, 2003.

[75] S. Zhou, G. R. Ganger, and P. A. Steenkiste, "Location-based Node IDs: Enabling Explicit Locality in DHTs," Carnegie Mellon University, Tech. Rep., 2003.

[76] S. A. Weil *et al.*, "CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data," in *ACM/IEEE SC*, 2006.

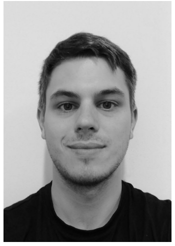
[77] P.-J. Maenhaut *et al.*, "A Dynamic Tenant-Defined Storage System for Efficient Resource Management in Cloud Applications," *Journal of Network and Computer Applications*, 2017.

[78] L. Rupprecht *et al.*, "SwiftAnalytics: Optimizing Object Storage for Big Data Analytics," in *IEEE IC2E*, 2017.

[79] R. Agrawal *et al.*, "Hippocratic Databases," in *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 2002.

[80] A. De Oliveira *et al.*, "Monitoring Personal Data Transfers in the Cloud," in *IEEE CloudCom*, 2013.

Martin Henze received the Diploma (equiv. M.Sc.) and PhD degrees in Computer Science from RWTH Aachen University. He is a post-doctoral researcher at the Fraunhofer Institute for Communication, Information Processing and Ergonomics FKIE, Germany. His research interests lie primarily in the area of security and privacy in large-scale communication systems, recently especially focusing on security challenges in the industrial and energy sectors.



Roman Matzutt received the B.Sc. and M.Sc. degrees in Computer Science from RWTH Aachen University. He is a researcher at the Chair of Communication and Distributed Systems (COMSYS) at RWTH Aachen University. His research focuses on the challenges and opportunities of accountable and distributed data ledgers, especially those based on blockchain technology, and means allowing users to express their individual privacy demands against Internet services.

Jens Hiller received the B.Sc. and M.Sc. degrees in Computer Science from RWTH Aachen University. He is a researcher at the Chair of Communication and Distributed Systems (COMSYS) at RWTH Aachen University, Germany. His research focuses on efficient secure communication including improvements for today's predominant security protocols as well as mechanisms for secure communication in the Internet of Things.

Erik Mühmer received the B.Sc. and M.Sc. degrees in Computer Science from RWTH Aachen University. He was a science assistant at the Chair of Communication and Distributed Systems (COMSYS) at RWTH Aachen University. Since 2017 he is a researcher and Ph.D. student at the Chair of Operations Research at RWTH Aachen University. His research interest lies in operations research with a focus on scheduling and robustness.

Jan Henrik Ziegeldorf received the Diploma (equiv. M.Sc.) and PhD degrees in Computer Science from RWTH Aachen University. He is a post-doctoral researcher at the Chair of Communication and Distributed Systems (COMSYS) at RWTH Aachen University, Germany. His research focuses on secure computations and their application in practical privacy-preserving systems, e.g., for digital currencies and machine learning.

Johannes van der Giet received the B.Sc. and M.Sc. degrees in Computer Science from RWTH Aachen University. He recently graduated from RWTH Aachen University and is now working as a software engineer for autonomous driving at Daimler Research & Development. His research interests include distributed systems as well as software testing and verification

Klaus Wehrle received the Diploma (equiv. M.Sc.) and PhD degrees from University of Karlsruhe (now KIT), both with honors. He is full professor of Computer Science and Head of the Chair of Communication and Distributed Systems (COMSYS) at RWTH Aachen University. His research interests include (but are not limited to) engineering of networking protocols, (formal) methods for protocol engineering and network analysis, reliable communication software, and all operating system issues of networking.