Project no.	732505
Project acronym:	LightKone
Project title:	*Lightweight computation for networks at the edge*

D6.2: New concepts for heavy edge computing

Deliverable no.:	D6.2
Title:	New concepts for heavy edge computing
Due date of deliverable:	January 15, 2019
Actual submission date:	January 15, 2019
Lead contributor:	TUKL
Revision:	2.0
Dissemination level:	PU
Start date of project:	January 1, 2017
Duration:	36 months

This project has received funding from the H2020 Programme of the European Union

Revision Information:

Date	Ver	Change	Responsible
Jun 30, 2018	1.0	1st version of this deliverable	TUKL
Jan 15, 2019	2.0	2nd version of this deliverable for resubmission	TUKL

Revision information is available in the private repository https://github.com/LightKone/WP6.

Contributors:

Contributor	Institution
Annette Bieniusa	TUKL
Deepthi Akkoorath	TUKL
Peter Zeller	TUKL
Mathias Weber	TUKL
Albert Schimpf	TUKL
Server Khalilov	TUKL
Dilan Shaminda	TUKL
Nuno Preguiça	NOVA
Roger Pueyo Centelles	UPC
Felix Freitag	UPC
Leandro Navarro	UPC
Roc Messeguer	UPC
Ilyas Toumlilt	Sorbonne Universite
Saalik Hatia	Sorbonne Universite
Dimitrios Vasilas	Scality
Marc Shapiro	Sorbonne Universite / Inria

Contents

1 Executive summary

Heavy-edge computing subsumes scenarios where the communication occurs primarily between devices from two different categories. Following the classification presented in the reference architecture in D3.1, this subsumes setups with Fat vs Thin, Fit vs Thin, or Fat vs Fit devices. Fat devices, for example, include cloud-based server deployments; thin devices comprise IoT and mobile devices with substantially less capacities; fit devices fall into the area in-between (e.g. routers). Based on their different computational power, storage capacity, etc., the basic communication patterns typically follow a client/multi-server setup. The heterogenous distribution of capacities in heavy-edge systems require an according distribution of tasks within the system. The major topic in this deliverable are distributed storage schemes for heavy-edge scenarios that allow for *partial replication*. The challenge hereby is to enable an efficient replication scheme that provides the consistency guarantees such as transactional causal consistency also at the client.

The core contributions of this deliverable can be classified into four categories:

- EdgeAnt and WebCure target partial replication on fit and thin devices with Antidote running on a fat device acting as a server.
- Our work on different aspects of consistency under high availability has continued in the area of access control and the Antidote Query Language. Further, we are working on an implementation of strong consistency mechanisms for Antidote and Partial-Order-Restrictions Consistency, both providing mechanisms to implement the Just-Right-Consistency paradigm (cf. D4.1.).
- We further report on related extensions, adaptations, and maintenance tasks on AntidoteDB and related software artifacts that are affiliated with the Lightkone Reference Architecture (LiRA).
- We finally present the reports on the design and implementation of the industrial use cases on implementing a monitoring system for the Guifi network (UPC) and the Proteus framework for federated metadata search in cross-cloud indices (Scality).

1.1 Project Milestones

D6.2. reports on the results achieved for Milestone MS4 targeted for month 18. Milestone *MS4: Heavy edge applications are successful* subsumes the design and implementation of the applications and the realisation of the needed infrastructure. The description of work breaks this milestone into the following subtasks:

- Realisation of industrial applications by partners Scality and UPC.
- Realisation of selected concepts for heavy-edge computation on top of AntidoteDB to make them available to the scientific community and allow for open evaluation by the academic partners.

In D6.1., the plans for the industrial use cases by Scality and UPC have been introduced. For D6.2., we report in Section 5.1 on the prototypical implementation of the monitoring application for the Guifi.net community network. The second industrial use case is a cross-cloud index for Scality's federated metadata search. For D6.2., the use case has been refined and detailed; Section 5.2 presents the design of the Proteus framework and an implementation plan.

The development of the use case applications has already lead to several extensions and improvements on the AntidoteDB platform, including the development of a REST API, a Go client, extensions to the API for setup and support for live monitoring. Further, we improved accessibility of AntidoteDB by providing a revised webpage, better documentation, different Docker images and interactive tutorials with the Antidote Playground. To facilitate the development of new features, several components are undergoing refactoring such as the test system, the CRDT library and the storage back-end. A detailed report is given in Section 4.

As of month 18, we argue that the milestone MS4 has been reached by 80%, given the preliminary status of the Scality use case.

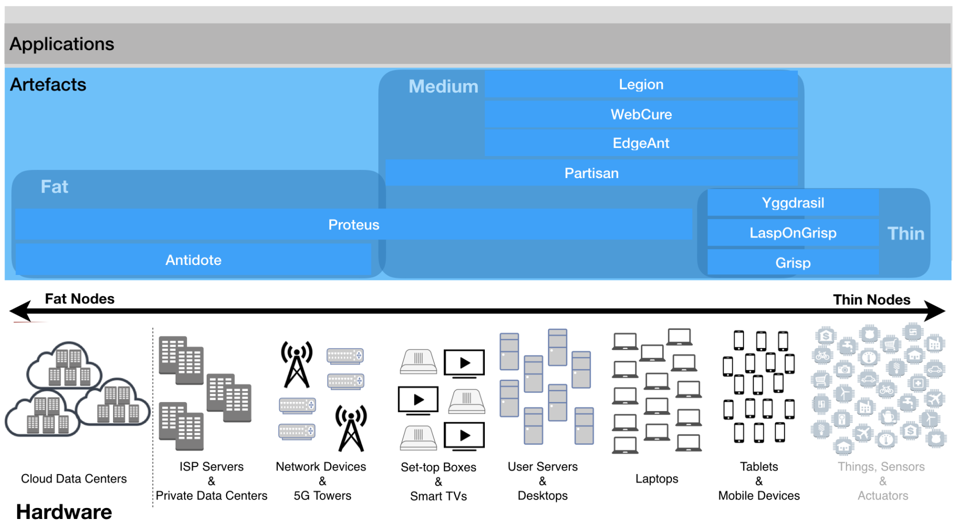


Figure 2.1: LightKone Reference Architecture: Architecture View.

2 Introduction

2.1 Reference architecture

The Lightkone Reference Architecture, presented in D3.1, categorizes different forms of edge computing that span a spectrum from light to heavy edge (see Fig. 2.1). WP6 targets heavy-edge computing where the communication occurs primarily between devices from two different categories. As outlined in the description of work, WP6 deals primarily with storage solutions for heavy-edge scenarios. In such a setting, the more powerful and reliable "fat" nodes, such as cloud or private data centers, typically are responsible for expensive computational tasks and persistent storage of associated data. To prevent unavailability in cases of node failure or network partition, we want to enable "medium" nodes placed at the edge to continue operating using locally cached data. The challenge hereby is to prevent or resolve conflicts on read and write access, by providing appropriate consistency guarantees and prevent data loss.

The work in WP6 is centered around with the following four artifacts:

- **AntidoteDB** is a geo-replicated cloud database based on the principle of synchronization-free programming. It offers a CRDT data model with highly-available transactions, and allows to combine different consistency guarantees that help to maintain typical application invariants. AntidoteDB supports both replication of data across different datacenters and sharding of the data space within a data centre. Application servers can interact with single Antidote nodes via different client libraries.

- **EdgeAnt** is a client-cache storage that ensures the same consistency and convergence properties as AntidoteDB while supporting partial replication on client side in a fault-tolerant way. EdgeAnt instances are expected to run on medium nodes interacting with a deployment of AntidoteDB as a back-end storage on the cloud.

- Similar to EdgeAnt, the **WebCure** framework provides client-side partial replication, but targeting web applications. It supports the development of progressive web applications through transparent interaction with an offline cache. It also relies on AntidoteDB as corresponding cloud database.
- **Proteus** is a geo-distributed system for analytics computations on federated data stores. It maintains materialized views and performs stateful data-flow computations. Proteus can support any combination of sharding, partial or full replication, and federation of data stores, thus enabling deployment on both fat and medium nodes.

In the first deliverable D6.1 for WP6, we outlined the work plan for WP6. We motivated the requirement for partial replication on client-side and highlighted the limitations of classical client-server interactions for geo-replicated cloud datastores, such as dynamic access control policies, query processing, and verification of application invariants. The report D6.2 now presents a supplementary discussion for state-of-the-art in partial replication, hybrid consistency for applications on geo-replicated datastores, and multi-cloud query federation.

We further report on the progress in the period M13-M18 for WP6. The focus hereby is on the four core artifacts, AntidoteDB, EdgeAnt, WebCure, and Proteus, and their exploitation in the industry partners' use cases.

2.2 Relation with other WPs

The work described in D6.2. has strong relations to WP2 where the requirements for the industrial use cases were introduced and formalized. The models have been refined and in parts already implemented as prototypes. The reference architecture developed in WP3 puts the heavy-edge scenario in context and describes the software artifacts and use cases in a broader context. WP6 further benefits from the development and improvement of CRDTs and communication protocols reported for in WP3. The results will lead to more efficient storage and bandwidth usage for the heavy-edge artifacts and feedback into the development in WP6.

The Just-Right-Consistency approach with corresponding verification tools, Repliss and CEC, that has been developed in the context of WP4, is the foundation for the semantics of AntidoteDB and its derivatives. The constraints indicated by our theoretical results on consistency in edge computing guided us in developing extensions such as support for strongly consistent transactions. The evaluation of the use cases will be targeted in WP7 and will feedback into the further development of the software artifacts for WP6.

2.3 Summary of Deliverable Revision

This deliverable has been revised to incorporate comments and modifications requested by the European Commission Reviewers. The main changes made to the deliverable are as follows:

- We revised the structure of the deliverable. In particular, we distinguish between core results relating to Lightkone's reference architecture and exploratory work beyond the project's artifacts.

- We have contextualized the results reported in this deliverable in relation to the Lightkone Reference Architecture.
- We have extended the state-of-the-art discussion to refer to additional relevant works, and to clarify the novelty of the work presented here.
- We evaluate the project milestones and give a detailed outline of the planned work for the last year of the project.

2.4 Outline of this deliverable

Section 2 gave a placement of the work performed in WP6 with respect to the Lightkone Reference Architecture and described the relation with the other WPs.

Section 3 presents the progress on the tasks relating to D6.2., addressing the challenges of heavy-edge scenarios. In particular, this subsumes different extensions to AntidoteDB that address challenges of developing highly-available, scalable and maintainable applications relying on a geo-replicated cloud datastore.

Section 4 provides references to the code repositories and documentation of AntidoteDB, related artifacts and applications. It further summarizes the improvements and changes done in the report period.

Section 5 describes the design, architecture, and implementation for the two use case applications of WP6.

Section 6 summarizes state-of-the-art for the core results of D6.2. and discusses how we improve on them.

Section 7 discusses additional exploratory research work that has been produced by partners of the Lightkone consortium.

Section 8 reports the scientific publications and dissemination activities that related to WP6 during the report period.

3 Progress and Plan

This deliverable reports on our progress and results for heavy-edge computing that we have achieved between M13 and M18 of the project. Most of the works described here were already motivated and introduced in D6.1., but have been significantly extended and advanced.

3.1 Partial replication

(a) EdgeAnt: Partial replication at client-side

As we described in the previous deliverable D6.1, the Swiftcloud approach [48] extends the data center causal consistency guarantees to the client local storage. The Swiftcloud system provides guarantees and techniques such as Read-Your-Writes, partial replication, K-stability, monotonic operations and small metadata design for tracking causality.
Based on the previous work, we implemented EdgeAnt, a client cache storage wish ensures the same consistency, convergence and fault-tolerance properties of Swiftcloud, using Antidote DB as a back-end storage, a rich API model and data/computation placement flexibility.

Design and requirements overview We consider a system model composed of a small set of powerful and geo-replicated Data Centers running Antidote DB (as described in D6.1), and a large set of limited resources clients.
Each DC hosts a full replica of data, and DCs are connected in a peer-to-peer manner. Antidote uses Cure protocol replication and its storage is operation-based which requires some protocol adaptation at clients partial-replica. DC can fail and recover from its persistent storage.
Clients stores a small and partial replica of the data, called there interest set, thus, an operation achieves high availability when the requested object is cached, but needs a remote communication when the object is missing in the local cache. Each EdgeAnt client is connected to a single DC, clients do not communicate directly. A client may disconnect, make offline local updates, than reconnect to its original DC or another one.
As in Swiftcloud, EdgeAnt decouples metadata design separating tracking causality, which is done using vector clocks in the DC side, and unique identification, based on scalar timestamps assigned in the client side.
Thanks to this design, and K-stability, metadata remains small and size-bounded in the number of DCs.

Client API EdgeAnt is a simple extension to the edge, where applications can interact with EdgeAnt the same way they could interact with Antidote, using its Erlang Protocol Buffer interface. The application first starts the transaction, then read/update one/multiple objects, and finally commit the transaction.

Transactions protocol When the EdgeAnt client first connect to the DC, it's assigned a global unique identifier composed of a scalar id and original DC id, this identifier will be attached to all transactions to ensure the "Only applied once" property, especially in the case when the client moves and applies its updates to another DC. Each operation will be then assigned a vector clock timestamp with respect to the Cure protocol. EdgeAnt client first connection also caches its initial state with the object from its first interest set, as explained in D6.1. The interest set of object keys is dynamically updated by the client, and is also stored in the DC side which will send back K-stable updates for its object.

Local updates are ordered using a scalar timestamp, the Commit Protocol (from client to DC) sends clients local commits to its connected DC, in background. In the DC side, received updates are applied with respect to clients timestamps order, causal dependencies (they can be client internal or external, the DC can report missing dependencies to the clients) and the aforementioned idempotence property. Then the geo-replication is done with respect to K-stability and Cure's stabilization protocol. Finally, the DC answers back to the client with the assigned vector clock, if this answer is not received, the client raises a DC failure and switches to another DC.

Each connected client has an EdgeAnt Session in the DC side, this process maintains not only the interest set but also the last known snapshot vector used by the client. Periodically, the DC will send over this session channel, a causal stream of updates. This notification update consists of a log of updates to the objects of the interest set, that are between the last know snapshot vector and the new one. This log can be empty, as we want also to notify the client when the snapshot vector changes due to external dependencies, and avoid causal gaps.

Moving computation Client computation resources can be poor and limited, although being resource-friendly and metadata lightweight, in EdgeAnt, some heavy operation can be done faster using the DC power. We are currently exploring a hybrid model where we can move computation from the client to the server in the heavy jobs case. This raises some interesting challenges like preserving the causal state of the client, handling updates and scheduling operations.

(b) WebCure: Partial replication for web applications

Despite improvements in connectivity for mobile devices by service providers, such devices are still subject to periods of disconnection. At the same time, users have higher expectations regarding the availability of applications: They want to interact with an app even when the device is (temporarily) offline. To provide an offline mode, apps need to cache the required data on the client machine. In addition, updates need to be recorded and forwarded to the server once a connection is re-established. This idea has led to many ad-hoc solutions that often don't provide well-defined consistency guarantees.

Web application running on client side are a typical scenario in which client-side replication is desirable. In current work, we are therefore developing a framework, named WebCure, for client-side partial replication in web applications. The framework uses Antidote as a cloud data store and supports client-side replication on selected data objects. It consists of three parts:

- A *client-side data store* is required to maintain the (partial) replica of the data that is relevant for the user. It maintains both the data that has been received from the cloud storage server and the updates that have been executed by the client, but have not been delivered to the cloud store, yet.

- A *service worker* acts as a proxy on client-side. It intercepts all calls to the cloud store. While the client is offline, it forwards the requests to the client-side data store.

- A *cloud storage server* that maintains data shared between different clients. For our framework, we chose Antidote as the cloud storage server. CRDTs are a very appropriate choice as data model since they provide a well-defined semantics for concurrent updates. Updates that are executed while a client is offline are concurrent with all other updates happening between the last retrieval from the cloud storage and the next connection and synchronization of the client.

As discussed in D6.1., we have shown with the Swiftcloud system [48] how to provide a partial replication scheme that extends the causal consistency of the cloud storage to client-side. Updates have to be recorded on the client to be forwarded to the server in case of connectivity. When reading data, we need to apply these operations on the last version received from the server to provide session guarantees such as Read-Your-Write and monotonic reads/writes. On the other hand, maintaining just a most recent version of a CRDT at the client is problematic. Differences in the cloud storage part of Swiftcloud and Antidote require adaptations of the protocols. Antidote is using an operation-based technique that is more efficient, but requires causality when deploying messages. In contrast, Swiftcloud employed a hybrid CRDT representation that enabled both operation-based and state-based convergence. We are currently adapting the Swiftcloud protocols and extend the Antidote API to support also the forwarding of meta-data to the client that is required for the CRDT operations.

In contrast to EdgeAnt, which is a feasibility study for edge-replication on Antidote, WebCure is motivated by the requirements and technology of web applications. WebCure's focus is on support for offline computation. Design decisions and heuristics for maintaining the offline cache are derived from this use case.

3.2 Challenges in Consistency

(a) Access control for weakly-consistent data stores

As we described in the previous deliverable D6.1., protecting sensitive data in applications on weakly consistent data stores is a challenge if the access control policies can change dynamically. The programmer might have an intuitive understanding of how access control should work:

> After revoking a right, all operation requiring this right cannot be executed until the right is explicitly granted again.

In setting where operations on data and policies happen sequentially, the semantics are based on the order in which operations are executed. However, in a system where updates on data and policies can happen concurrently, it is not straightforward how to resolve conflicting accesses. As we have shown in prior work [46], replicated data types allow to encode conflict solving strategies into the data type semantics itself. We have developed a Policy CRDT that resolves conflicting update operations on a policy in a safe, restricting way. Further, we have shown how causal consistency and transactional data access allows to implement an access control semantics that reflects the programmer's intuition about access control without requiring serialization of data and policy access.

Our work on access control has meanwhile progressed both in theoretical and practical aspects. In the previous report, we considered a simplistic access control model where each operation was associated with an access control policy. Such a policy states whether an (abstract) subject is allowed to execute a specific operation on some object. In practice, more advanced authorization models have been established. In attribute-based access control, attributes of the subject, the objects, and the operations are taken into account when evaluating the access control policy. This approach allows to implement a wide range of access control policies, including also approaches such as role-based access control (by adding a role attribute to a subject).

To validate our model of attribute-based access control on weakly consistent data stores, we have derived a machine-checked proof of the correctness using Isabelle/HOL. Further, we implemented a prototype of this attribute-based AC framework based on Antidote's Java client. In preliminary evaluations, we ran micro-benchmarks on our local server. Our prototype shows high scalability in work loads with a moderate overhead of approximately 30% for the evaluation of the access control policies. In a next step, we are planning to evaluate our system on AWS. For the corresponding evaluation plan, we refer to D7.1.

(b) Antidote Query Language

We are currently working on Antidote Query language (AQL) to extend its support for queries over large collections of data. In that effort, we have added support for indexes, range queries and server-side operations. To implement these features, we came across a number of challenges that we detail in the following. We start this section by providing an overview of the goals of AQL.

AQL overview AQL is a SQL interface for AntidoteDB. As other SQL interfaces for NoSQL databases, the main goal of AQL is to allow programmers to access the database through the familiar SQL interface. We currently support only a subset of SQL, with one of the major limitations being the fact that joins are currently not supported.

What is unique in AntidoteDB, when compared to SQL interfaces for weakly consistent databases, is its support for SQL invariants and a concurrency-aware schema definition. These features were discussed in D6.1. During this reporting period, we have been improving the implementation of these features by minimizing the meta-data manipulated during operation execution.

Building indexes on weakly consistent data In classical database systems, tables are normally stored in order of their primary key to allow fast look-up and scan operations based on that key. When an application needs frequent data access based on values that are not keys, it is common to create secondary indexes to speed-up those operations. A secondary index for a table $T = (id, col_1, \ldots, col_N)$ consists of a mapping structure $\{value_i \rightarrow \{id_i, \ldots, id_n\}\}$, where $value_i \in T.col_x$, for all distinct values of $T.col_x$, and $\{id_i, \ldots, id_n\} = \{r.id_j \mid \exists r \in T, r.col_x = value_i\}$. With this structure it is possible to search for rows based on the indexed column col_x, without actually traversing the table content.

AntidoteDB uses a key-value store as the backend store. Currently, each key points to a CRDT Map that stores the content of a table. Each key of the map represents a primary key of the table and points to another map that stores the remaining column values. Each column value is itself a CRDT.

As in traditional databases, this design poses limitations for executing queries based on non-key values, since it is necessary to access the content of each row to check if it matches the query filter. We have added support for secondary indexes in AntidoteDB to solve that issue.

To maintain the index, we use a mapping structure as described before, and for each operation on the indexed table, we update the index as follows: on inserts, we add the new row's primary key to the set of identifiers of the corresponding indexed column value; for removes, we remove the identifier; and for updates, we remove the identifier from the old entry and add the identifier to the new entry. However, this simple strategy is unable to maintain the integrity of the indexed table under concurrent executions.

Consider that two concurrent operations add a new row (id_i, v_1) and (id_i, v_2) to table $T = (id, c_{idx})$, with an index *idx* on column c_{idx}. To update the index, each operation adds new entry $v_1 \rightarrow \{id_i\}$ and $v_2 \rightarrow \{id_i\}$ on index *idx*. Since the two operations on table T conflict (because they use the same identifier), a conflict resolution for column c_{idx} is applied, eliminating v_1 or v_2. However, since each operation created a distinct entry on the index, both values are preserved, leaving the index inconsistent.

To solve this issue, we store in the index the set of primary keys and the corresponding indexed value's CRDT, $Indirection = (id_i, CRDT(c_{idx_i}))$. With this auxiliary data, we are able to reproduce the conflict resolution rule on the index and discard the element that was lost during the conflict resolution. We explain the scenario with an example. When the first operation in the previous example is applied on the index, a new entry $(v_1 \rightarrow id_i)$ is created and the pair $(id_i, CRDT(c_{idx_i}))$ is added to *Indirection*. When the second operation is applied, that primary key has been already added to the index, and thus a conflict arises when adding the pair $(v_2 \rightarrow id_i\})$ to *Indirection*. We reapply the operation on the CRDT associated to id_i to get the latest value for c_{idx_i}. Now, we remove all occurrences of the primary key in the index and create (or update) the entry of the latest value c_{idx_i}. This ensures that the index is consistent at all times.

With secondary indexes, we are able to execute `WHERE` clauses more efficiently when the filter condition contains columns that are non primary keys and an index is available. We also added support for inequality operators, such as $\leq$, $<$, $>$, and $\geq$. Range queries are supported by traversing the indexes in-order (primary/secondary). It is not recommended to execute range queries over non-indexed columns.

Server-side operations To execute queries, AQL needs to fetch the data from Antidote before applying the query filters. This poses an overhead on the network for transferring large amounts of data that might end up being discarded by the AQL query processor. For instance, to execute a query statement like `SELECT * FROM T WHERE T.id = x`, it would be necessary to fetch the whole table. To avoid transferring unnecessary data, we are extending Antidote with server-side queries. Server-side queries allow simple query operations over individual CRDT objects which allows, for instance, to filter table rows before passing the data to AQL.

This mechanism would allow a query optimizer to request partial views of a CRDT to reduce the amount of data that has to be transferred to execute a query. For example, consider a query that joins two tables, but discards all values that are older than a certain threshold. With server-side operations it is possible to filter the rows that do not meet the threshold before sending the response to AQL.

To support this feature it is necessary to extend the interface of CRDT objects to return a partial view of their state and extend the Antidote interface to support direct queries on objects. This feature is currently being developed.

REST API We have added support for issuing AQL statements through a REST interface. The interface exports a single endpoint, with the statement being submitted in the body of the request. The results of executing the statement is returned in the body of the reply. Currently this REST API is not integrated with the main AntidoteDB API, but we plan to integrate it when AQL is integrated in the master.

(c) Strong Consistency for Antidote

When designing a distributed application, the semantics for accessing the data maintained by the application is defined by the consistency model of the respective data store. The choice of consistency model follows a certain trade-off: synchronous models, such as linearizability or serializability, impose a global order on operations and thus a simplified means of reasoning about application semantics; asynchronous ones, such as eventual consistency or causal consistency, offer low latency responses and tolerate partition, but are more difficult to understand. With the Just-Right Consistency (JRC) approach that we introduced in D4.1., we aim to minimize the amount of synchronization required to ensure the applications invariants. JRC builds upon common invariant-preserving programming patterns. The patterns “ordered updates” and “atomic grouping” are compatible with concurrent and asynchronous updates. The “precondition check”, however, requires synchronization only in certain cases: Under network partitions, the state might change at a remote replica, thus falsifying the local precondition check. If an analysis identifies transactions, which could invalidate application invariants, we require synchronization mechanisms from the datastore to preserve these invariants. This means that the datastore should provide a interface for highly available and partition tolerant (AP) transactions and one for strongly consistent and partition tolerant (CP) transactions.

As described in D4.1, Antidote provides AP transactions on CRDT data objects with transactional causal consistency as consistency model. In addition, bounded counter CRDTs can be used to maintain invariants on counters that are guaranteed to not cross specified values. To fully support the JRC approach, we are currently developing an extension to the Antidote API that provides distributed locks. Transactions can take as parameters a set of locks that are required for executing the transaction safely. These CP transactions are guaranteed to execute causally consistent with all other transactions and linearizably with respect to other CP transactions that synchronize on the same locks. It thus follows the RedBlue consistency model [30] distinguishing between fast eventually consistent operations (blue) and (potentially more costly) strongly consistent operations (red). In a first prototype, we have implemented simple locks that are transferred upon request between Antidote clusters. The on-going implementation can be found in a branch of Antidote’s Github repository [1]. We are planning to investigate and compare further locking strategies such as multi-level locks to provide a more fine-grained approach.

[1] https://github.com/SyncFree/antidote/tree/strong_consistency

3.3 Plan for final year

As anticipated in the description of work, the design and implementation of the industrial use cases by UPC and Scality have provided valuable feedback for the heavy-edge artifacts of the Lightkone Reference Architecture.

In the final year, the agenda for WP6 comprises the following tasks:

- Scality will finish the ongoing implementation of the Proteus framework and make it accessible as open-source artifact.
- To support UPC's evaluation of the monitoring application using AntidoteDB on several hardware platforms as described in D7.2., NOVA, TUKL and SU will provide variants of AntidoteDB targeting resource-constraint environments and deployment support for different network topologies. To this end, SU will also finalize the development of EdgeAnt and provide a packaged, deployable version of the corresponding software.
- TUKL and SU will provide an improved implementation of AntidoteDB's backend that allows for a more efficient handling of persistent state.
- INESC TEC will implement a Tagged Causal Broadcast middleware using a novel algorithm. This middleware will be evaluated as an alternative for inter-DC communication in AntidoteDB.

We further plan to incorporate feedback from the implementation and evaluation of the use cases as described in D7.2. as needed for a successful completion of the industrial use cases. This will subsume on-going work for the Antidote Query Language and maintenance of AntidoteDB's codebase.

4 Software Deliverables

The following software artifacts from this report are publicly available:

- WebCure‡
- EdgeAnt‡
- Access Control (ACGreGate)‡
- Antidote Query Language (AQL): https://github.com/JPDSousa/AQL
- REST API for Antidote : https://github.com/LightKone/antidote-rest-server
- Antidote Go client : https://github.com/AntidoteDB/antidote-go-client
- Antidote with strong consistency : https://github.com/SyncFree/antidote/tree/strongconsistency
- UPC Monitoring App: https://lightkone.guifi.net/lightkone/uc-monitor-go-test
- Calendar App: https://github.com/AntidoteDB/calender-app

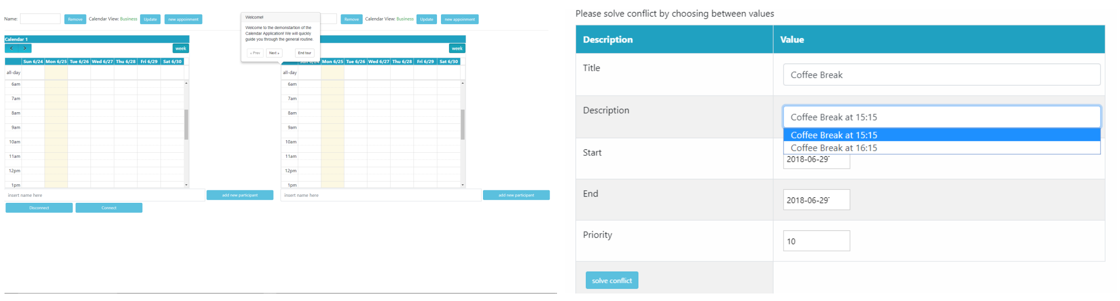
‡These softwares are potential valorization targets. Access to the code can be obtained upon request.

In the following, we summarize the work on the software deliverables for WP6 in M12-M18.

(a) Antidote Playground

The semantics of concurrent access to shared data can be challenging to understand for programmers. In contrast to eventually consistent replicated key-value stores with few guarantees, CRDTs have a well-defined semantics that provides convergence of concurrent updates. Nevertheless, it can be intimidating for programmers with little prior exposure to CRDT theory to understand their semantics. Often, it is also unclear how to derive a data model that benefits from a data-type driven approach to replication and conflict resolution. We therefore implemented a web application to interactively explore their semantics. The Antidote playground allows users to simulate concurrent operations on shared data by explicitly introducing and healing network partitions. In its current form, it features an interactive shell where objects under keys can be read and updated using a dedicated language for the interaction. Further, we developed a shared calendar with safe conflict detection and resolution (Figure 4.1). By performing concurrent modifications on the same calendar entry, the user see the usage of AW-Sets and Multi-Value Registers in an application.

Technical setup The Antidote Playground is deployed on a local cluster of Docker containers, a machine containing 16cores Intel(R) Xeon(R) CPU E5-2609 v2 @ 2.50GHz, 64GB memory and 512Go SSD hard drive. Currently, the cluster is only located in Paris; we plan to geo-replicate it to two more locations (Lisbon and Kaiserslautern) in the future. On the software side, the cluster is running Ubuntu 16.04, Linux 4.4.0, Docker 17.3



Please solve conflict by choosing between values

Description	Value
Title	Coffee Break
Description	Coffee Break at 15:15 Coffee Break at 15:15 Coffee Break at 16:15
Start	2018-06-29'
End	2018-06-29'
Priority	10

Figure 4.1: A shared calendar using Antidote as a backend. Concurrent updates on Multi-value Registers such as the description of an entry require user intervention; concurrent updates on other data types are handled with according to the respective CRDT semantics.

and Erlang 19.3. Each instance of the Antidote Playground is composed of 3 Docker instances running the last stable Antidote docker image. The Antidote instances are inter-connected via local network (on ports 8087/8088/8089) with a simulated latency (fixed to 1ms by default). The Antidote Playground API offers a Java interface to manage network partitions (GET partition status, ADD partition and DELETE partition by id). The Antidote Playground web application is also contained in a Docker instance that is HTTP-opened to the world through 3000 port.
Currently, the Antidote Web Shell application is publicly running on the cluster and is accessible from http://antidote.lip6.fr:3000/. This application offers a number of simple Shell commands to manipulate CRDT AW sets, counters, and simulate network partitions.

The Antidote playground will further be made available on the Antidote webpage[2].

(b) Improvements to CRDT library

The CRDT library was improved to be easier to use by third parties. In particular, we improved the documentation and removed experimental data types from the master release. The removed experimental CRDTs are:

- The integer, because we would prefer a semantics, where an assignment of a value is not changed by concurrent increments. However, an efficient design for this semantics most likely requires support for generating actor identifiers outside of the library (e.g. in Antidote), for which we have yet to find an elegant solution.

- The replicated growable array (RGA), because it contained bugs and inefficiencies. We are currently implementing better CRDTs for storing lists.

- The add-wins map was removed, because it does not implement garbage collection of meta-data. The alternative recursive-resets map is therefore more efficient. Currently the recursive-reset map has some deficiencies: not all CRDTs can be used as values in the map and the map cannot distinguish entries with an empty value from missing entries. We are planning to improve the recursive-resets map to have the same semantics as the old add-wins map.

[2] http://antidotedb.eu

(c) Specification of Antidote back-end interface

Antidote stores its operations sequentially in a durable log. It allows asynchronous, sequential writes of multi-objects transactions and atomic writes which is efficient for operation based CRDTs. However, this creates the following problems; with time the journal grows without bound and recovery from crash gets slower as it implies reading the entire log in order to restore its in-memory state.

We are currently redesigning the durable store. We keep the log, but combine it with durable storage of checkpointed object versions. This way we keep the advantages of the log, but we can truncate information that is either obsolete or redundant with the Checkpoint Store. The overarching objective of the specification is to guarantee that every object, and every live version of an object is durable. The fact that an object or object version is live is a property defined by the application. Furthermore we gain efficiency by storing small updates with the journal and large states with the Checkpoint Store. Strictly speaking, only the journal is necessary; the Checkpoint Store allows us to truncate the journal and store materialized versions of an object, making it more efficient for edge devices. Therefore the back-end store design combines two parts:

- *Journal*: A sequentially written log of update operations. Each Antidote partition server has its own private journal.
- *Checkpoint Store*: A KV store of materialized object versions. The Checkpoint Store may be private or shared across Antidote shard servers.

Journal: The journal is a sequence of opaque records. It is write-once, read-many and is optimized for sequential access. Each record is identified by a *RecordId*, which is a unique record identifier, a monotonically increasing number assigned by the journal.

The journal has two special RecordIDs; the *highWatermark* makes the upper bound of records known to be durable, and *lowWatermark* the lower bound.

Checkpoint Store This is where Antidote store the checkpoints of object states. Our design stores write-once versioned objects. This implies that the Checkpoint Store supports creating new versions atomically, and listing an object's versions. Every object is identified by its key, and an object version is identified by a pair (Key, Timestamp) where Timestamp is a vector. We distinguish data and metadata by a secondary identifier SubID. The Checkpoint Store is a map of (Key, SubID, Timestamp) to blobs. The store can use SubID to optimize object data for large writes, versus metadata for small writes.

In order to rebuild a transactionally causally consistent snapshot, either because it is not cached in memory or on recovery after a crash, it requires to list the versions of a key. It must be the case that every live version can be reconstructed either from the Checkpoint Store, or from the journal and Checkpoint Store. Our further work will be to specify the corresponding invariants of the back-end store, adapt the materializer to this new specification, proving correctness and testing and deployment with an challenging application, a petabyte-scale edge file system.

(d) DC management API

An Antidote data center (DC) is a cluster of multiple Antidote nodes. An Antidote deployment can extends to several DCs where each DC maintains a full replica of the data stored. After starting several Antidote nodes, the user has to instruct them to form a DC and connect them to start data replication among the DCs. Previously, this DC connection operation was only possible by calling Erlang functions through the Erlang VM's RMI interface. Meanwhile, we provide a DC management API exposed over the protocol buffer interface. With the new API, users can create DCs by specifying the Antidote nodes that belongs to a DC and connect multiple DCs for replication using the same language client libraries that we provide for data access in Antidote. This has significantly simplified the task of setting up Antidote clusters and dynamically maintaining their state.

(e) Test system

Testing distributed systems is notoriously difficult, though many problematic situations can be prevented by simple unit tests [45]. To facilitate the testing of Antidote and its new extensions, we have refactored Antidote's test suites and provide now documentation and guidelines for developers to write tests.

Antidote features both unit tests for single modules via Erlang's eunit and system tests via the Common Test framework that help to test components and the entire system. The goal of the test suites is to define deterministic execution sequences that also include failure cases. To this end, we provide test libraries that allows to disconnect, kill, and restart nodes.

With the refactoring, we managed to reduce the execution time of the test suites from approximately 26 min to 18 min. The test coverage is now at 70% for the system tests. We hope to improve it further in the next time.

(f) Go client

The Go programming language developed by Google is "an open source programming language that makes it easy to build simple, reliable, and efficient software" [3]. Antidote offers now also a client library for Go which enables developers to use the Antidote data store from Go programs. Because Go is designed to be a systems programming language, we hope to open opportunities to use Antidote from performance constrained software such as software running in edge devices.

(g) REST API

Early in the development stages of the UPC Use Case (UC) it was noticed that having a language-independent interface to AntidoteDB would help in the process of bootstrapping the application code. This would apply not only to that particular case, but would also ease the adoption of AntidoteDB in other scenarios. Given the fact that most modern languages support interaction via HTTP (e.g. making GET calls), an HTTP/HTTPS Representational State Transfer (REST) Application Programming Interface (API) was developed, providing an additional interface for applications to interact with AntidoteDB.

At its current stage, the REST API provides the following operations for a subset of the Conflict-free Replicated Data Types (CRDTs) available in AntidoteDB:

- Counter
    - GET /counter/read/:bucket/:key

[3] https://golang.org/

  - GET/PUT /counter/increment/:bucket/:key/:amount?

- Integer
  - GET /integer/read/:bucket/:key
  - GET/PUT /integer/increment/:bucket/:key/:amount?
  - GET/PUT /integer/set/:bucket/:key/:value
- Set
  - GET /set/read/:bucket/:key
  - GET/PUT/POST /set/add/:bucket/:key/:elem
  - GET/DELETE /set/remove/:bucket/:key/:elem
- LWW Register
  - GET /register/read/:bucket/:key
  - GET/PUT /register/set/:bucket/:key/:value
- MV Register
  - GET /mvregister/read/:bucket/:key
  - GET/PUT /mvregister/set/:bucket/:key/:value

The API was implemented as a Node.js [4] server application, using the CoffeeScript [5] framework. Its source code is available at LightKone's GitHub public repository [6] and its also packaged and ready for usage at npm's repository [7].

The REST API server can be installed with npm:

```
$ npm install -g antidote-rest-server
```

By default, the REST API is run as a standalone server listening at *localhost*'s TCP port 3000 with the following command:

```
$ antidote-rest-server
```

Additional configuration options allow to indicate the location (address and port) of the AntidoteDB host to which the REST API provides the interface, the port where the REST server listens for connections, etc.

A quick start guide for the REST API is available at the repository wiki [8].

[4]Node.js - https://nodejs.org/es/

[5]CoffeeScript - https://coffeescript.org/

[6]AntidoteDB REST API server source code - https://github.com/LightKone/antidote-rest-server

[7]AntidoteDB REST API server package - https://www.npmjs.com/package/antidote-rest-server

[8]AntidoteDB REST API QuickStart - https://github.com/LightKone/antidote-rest-server/wiki/QuickStart

5 Industrial use cases

5.1 Monitoring Guifi.net community network

The distributed components of the Guifi.net monitoring application that performs the *monitoring servers* $\Leftrightarrow$ *network devices* mapping use AntidoteDB to store data and, indirectly, as a mechanism for coordination between the different servers. For a detailed description of the use case, we refer to deliverable D2.1.

In order to easily interact with AntidoteDB, the need for an interface that was programming language-independent arose at the early stages of the monitoring application development. This was also important in the context of Guifi.net, where contributors to the codebase may not be experts on AntidoteDB's implementation language (i.e., Erlang). For this reason, we developed the Hypertext Transfer Protocol (HTTP) REST API described in Section 4(g): to enable instances of the monitoring application to conveniently read/write data from/to AntidoteDB, but also to foster the bootstrapping of other applications in different scenarios.

A proof-of-concept monitoring application has been implemented, in the Go programmig language [9], and is available in a public repository [10]. It is organized in three main modules that perform different tasks, which are detailed below.

monitor-fetch: network description fetching and feeding to the shared database A hierarchical description of the whole Guifi.net network is available in the Extensible Markup Language (XML)-formatted file *cnml.xml*, which can be downloaded from the Guifi.net website. Besides this global *cnml.xml* file, we use also smaller files which describe smaller parts of the Guifi.net network (*bellvitge.xml* -a small mesh subnetwork-, *upc.xml* -a bigger mesh subnetwork- and *barcelona.cnml* -which includes the previous one-. Being the primary data source for this module a Community Network Markup Language (CNML) file, the *monitor-fetch* program parses the specified CNML file to filter only the required information and push it to AntidoteDB. There is a single *monitor-fetch* instance, and its writes and updates to AntidoteDB are considered to be authoritative.

Once parsed from the CNML file and dumped to AntidoteDB, the data is structured as follows:

guifi (bucket): The guifi bucket contains the lists of monitors and network devices to be monitored.

guifi (bucket) $\Rightarrow$ devices (set): The *devices* set in the *guifi* bucket is an array of strings, each string containing the numeric ID of a Guifi.net device. For example:

```
$ curl localhost:3000/set/read/guifi/devices
["2110","26932","38720","40605","40962","41175","42331","42626",
 "42627","42628","46654","46656","47103","48030","51580","57728",
 "59001","60415","64962","64963","64965","64966","65291","65720",
 "72843","73952","79715","81297","82096","82097","82098","82099",
 "82103","82104","82105","82111","83865","85877","87503","90228",
 "92032","92802","92803","92804","94210""94965"]
```

[9]The Go Programming Language - https://golang.org/

[10]https://lightkone.guifi.net/lightkone/uc-monitor-go-test/tree/antidote-rest

guifi (bucket) ⇒ monitors (set): The *monitor* set in the *guifi* bucket is an array of strings, each string containing the ID of a Guifi.net monitor. We distinguish the current monitors registered at the Guifi.net website from the monitoring servers used in our prototype by adding the prefix `a` to their ID. When a monitoring server starts, each of the monitoring servers registers with the system by adding its ID to this set. For example:

```
$ curl localhost:3000/set/read/guifi/monitors
  ["a45632","a87363", "21435", "41229"]
```

guifi (bucket) ⇒ checksum (LWW1 register): The *checksum* LWW register in the *guifi* bucket is a string containing the SHA256 checksum of the last CNML data fetched from the Guifi.net website and pushed to the database. For example:

```
$ curl localhost:3000/register/read/guifi/checksum
  35aaa826b841ed412897691bb1f50278d742ef9a76da9750a8ae509d3b01f8ee
```

device-i (bucket): The *device-i* bucket, where *i* is the numeric ID of a device in the *guifi* ⇒ *devices* set, contains the information about a specific Guifi.net device:

device-i (bucket) ⇒ ipv4s (set): The *ipv4s* set in the *device-i* bucket is an array of strings, each string containing an IPv4 address of the device. For example:

```
$ curl localhost:3000/set/read/device-26932/ipv4s
  ["10.139.37.226","172.25.40.188","172.25.40.189"]
```

device-i (bucket) ⇒ monitors (set): The *monitors* set in the *device-i* bucket is an array of strings, each string containing the ID of a monitor the device is assigned to (i.e. the ID of a monitor that is in charge of monitoring the device). For example:

```
$ curl localhost:3000/set/read/device-26932/monitors
  ["a45632","a47363", "21435"]
```

device-i (bucket) ⇒ graphserver (LWW register): The *graphserver* LWW register in the *device-i* bucket is a string containing the ID of a monitor the device is assigned to in the Guifi.net website (i.e., not automatically assigned by the monitoring application, but done manually on the Guifi.net website, and included in the CNML). For example:

```
$ curl localhost:3000/register/read/device-26932/graphserver
  71808
```

monitor-assign: network devices assignment among the different monitoring servers The *monitor-assign* modules that run at each monitoring server perform the distributed assignment of network nodes among the different servers. To do this, it relies on the network information already written to AntidoteDB by the *monitor-fetch* module. The assignments decided locally are written back to Antidote to the device-i (bucket) / monitor set. In our current WiP implementation we apply a random network node to server placement as assignment policy.

3) Monitor the network nodes: This component under WiP aims to support the nodes to monitoring server assignment by other policies, which as effect may increase the overall reliability or fulfill specific criteria of the monitoring system.

The next steps in the prototype development will interact with the results from the evaluation conducted in WP7 and the obtained feedback will consolidate the next version of the application design.

5.2 Building a cross-cloud index for Scality's federated metadata search

Scality's open source multi-cloud framework, Zenko [11], enables applications to transparently store and access data on multiple public cloud storage systems, including Microsoft Azure Blob Storage, Amazon S3 and Google Cloud, as well as private on-premise storage systems, using a single storage interface (an Amazon S3 compatible API). Zenko supports cross-cloud metadata search, enabling applications to retrieve data by performing queries on metadata attributes across multiple cloud namespaces.

Scality's use case aims at introducing a geo-distributed metadata search service as a replacement to the current implementation which supports cross-cloud metadata search by gathering and storing metadata attributes on a database placed on a single location. The new design aims at improving the search system's state and computation placement flexibility. We describe this use case in more detail in deliverable D2.2.

Proteus: A framework for making trade-offs in distributed querying Implementing distributed querying systems, such as Zenko's metadata search, is challenging as the problem has a large design space with multiple dimensions. Querying systems need to optimize different metrics, such as search latency, write latency, and search result freshness, which are often in tension. Therefore, no single design can be appropriate for all uses; design decisions depend on data distribution and replication schemes, as well as the requirements of each particular application.

Proteus is a geo-distributed framework for analytics computations on federated data stores. It maintains materialized views and performs stateful data-flow computations. Proteus can support any combination of sharding, partial or full replication, and federation of data stores. In the context of the project we will use an instantiation of Proteus in which materialized views are realized as secondary indexes and search result caches, and the framework can perform distributed query processing. Therefore, Proteus will be used as a distributed query processing framework for transparently extending geo-distributed data stores with querying capabilities. An earlier version of this work [43] has been introduced in D6.1.

Proteus is by design modular and flexible. It enables administrators to place data and computations according to SLA considerations. More specifically, querying system implemented using Proteus can make varying design choices about:

- **Search mechanism:** Queries can be processed either by scanning the dataset and filtering objects that match a given query, or by maintaining secondary indexes. These approaches create a trade-off between search latency and the storage and maintenance overhead of maintaining indexes. Additionally, search result caching can be used to speed up query processing.

- **Data and computation placement:** The system's state (indexes and caches) can be flexibly placed across a geo-distributed system architecture. The state placement strategy affects communication patterns for index maintenance and query processing. Fro example a secondary index can be partitioned in partial indexes, and these

[11] https://www.zenko.io/

indexes can be distributed across multiple locations or placed closer to the clients at the edge of the system.

- **Consistency model:** Secondary indexes can be either eventually-, causally- or strongly-consistent, according to the capabilities of the underlying data store.

By allowing flexibility across these dimensions Proteus can express multiple points in the design space of distributed query processing. As a result, Proteus can be tailored to address multiple different application characteristics and requirements.

Technically, Proteus runs a bidirectional data-flow computation, and maintains internal state that is used for processing queries. A write to the data store streams upwards through the data-flow graph, and incrementally updates indexing structures. Conversely, a query streams downwards through the data-flow graph, and is incrementally split into sub-queries that are processed, either using Proteus' internal state, or by querying the data store. Queries also update Proteus' internal state, for instance query results can be cached. The data-flow graph of Proteus is modular, and enables flexible placement of state or computation, across a geo-distributed system architecture.

System Architecture Proteus architecture is composed of software components called Query Processing Units (QPUs). QPUs act as microservices that perform primitive query processing tasks, such as indexing, scanning and caching.

Below we present the different types of QPUs in the system, their functionality and their interface.

Data Store QPU: The data store QPU works as a wrapper that exposes a common interface to the rest of the querying system independent of the underlying storage system.

An object stored in the data store is composed by a blob of data, a set of secondary attributes represented as a map (MDKey, MDValue), where MDKey and MDValue are binary values.

An object is identified by:

- Key: Unique object identifier
- Timestamp: Vector identifying the version of an object

The data store QPU exposes the following API:

- $listObject(Timestamp \quad ts_1, Timestamp \quad ts_2) : \quad StreamOfObjects$
  Creates a stream of all the object versions with $ts : ts_1 \leq ts < ts_2$. For each object, the result contains its key and metadata attributes.

- $getOperations(Timestamp \quad ts) : \quad StreamOfOperations$
  Creates a stream of the operations with $ts : ts_1 \leq ts$. This stream remains active until explicitly closed by either the sending or receiving end, and each new operation performed in the data store is sent through the stream.

Scan, Index, Cache and Dispatch QPU The remaining QPU types expose a common interface. This interface is used by clients for issuing search requests to the querying system as well as for communication between QPUs.

These QPU types expose the following API:

- $find(Query\quad q, Timestamp\quad ts_1, Timestamp\quad ts_2):\quad StreamOfResults$

  where a *Query* is a map of (MDKey, MDValue, MDValue), which represents a list of secondary attributes and attribute value ranges.

  Creates a stream of all the object versions that match the given predicate with $ts : ts_1 \leq ts < ts_2$.

Each type of QPU has a specialized implementation of this interface:

- Index QPUs maintain partial secondary indexes and process queries by performing index lookups. The index QPUs make use the data store QPU $getOperations$ API in order to receive updates performed to the data store and build secondary indexes.
- Scan QPUs use the data store QPU $listObject$ API in order to receive a stream of object stored in the data store and filter the objects that match a given query.
- Cache QPUs maintain a cache recent query results and respond to queries using their cache or forward queries to other QPUs using their $find$ interface.
- Dispatch QPUs process queries by decomposing them to sub-queries and forwarding them to other QPUs for processing, using their $find$ interface.

Proteus is used by deploying QPUs and interconnecting them in a network by configuring the communication between them. Different QPU network configuration can make different design choices about (1) the types of QPUs used, (2) their placement in the system, and (3) the communication patterns among QPUs and between QPUs and the underlying storage system. These design choices express different points in the design space of the problem of distributed query processing.

The query processing protocol works in a decentralized fashion. Given a query, the receiving QPU first determines if it can process it locally (by performing an index/cache lookup or a data store scan). Otherwise, the QPU decomposes the given query to sub-queries, forwards them to (some of) its neighbors, and then combines the received responses and responds. Each of the neighbor QPUs that receives a sub-query recursively runs the same protocol.

QPUs in Proteus use Antidote as a backend database for storing indexes as CRDTs. We are also currently studying designs for using Antidote to provide causality and atomicity guarantees in index maintenance.

Implementing multi-cloud metadata Search using Proteus We plan to implement a geo-distributed metadata search service in Zenko using the Proteus framework. The design consists of index QPUs organized as a hierarchical network that implements a geo-distributed weakly consistent index. The index is partitioned, and index partitions are distributed across different locations.

A geo-distributed querying system implemented using Proteus is depicted in Figure 5.1. A data store QPU is deployed locally at each backend cloud storage system and is responsible for propagating local updates to an index QPU that is also deployed on the same location for achieving low latency index maintenance. This requires that the clouds provide an event notification mechanism of some form, which the data store QPU

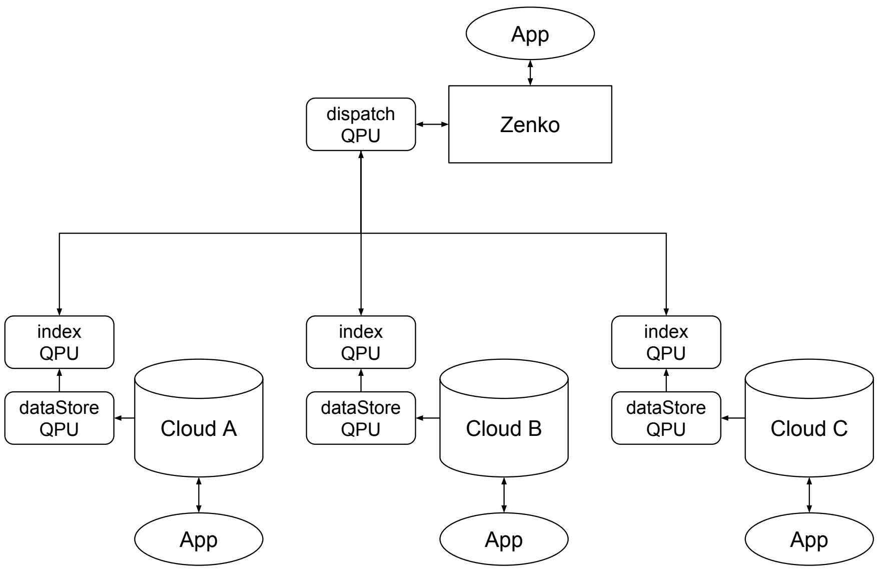


Figure 5.1: Zenko Modular Metadata Search Architecture .

can use to get notified for write operations issued directly to the storage systems. Writes performed through Zenko are eventually propagated to the backend storage systems and indexed through the same mechanism. Alternatively, an index QPU can be deployed along with Zenko and be responsible for indexing write operations performed directly through Zenko. Note that each index QPU shown in Figure 5.1 represents a sub-network of index QPUs implementing a distributed index. A dispatch QPU is deployed along with Zenko; when a search performed (through Zenko), the dispatch QPU forwards it to all index QPUs and then merges the retrieved results.

A querying system can be implemented using Proteus by deploying QPUs as Docker containers and providing each QPU with a simple configuration describing its behavior, its connections to neighboring QPU within the network, and its connection with the underlying storage system. Proteus then handles the communication among QPUs for performing query processing.

6 State of the art

In this section, we give a comprehensive overview on state-of-the-art for D6.2. In particular, this covers partial replication in highly-available systems, including several frameworks for web and mobile application development, and approaches for multi-cloud query federation.

Partial replication Data placement and replication factors are a critical aspect when building highly-available systems extending to edge devices. In contrast to caching systems, partial replication provides means for asynchronous updates on the edge device.

Achieving low latency for web-based applications is an on-going challenge for many web applications [1, 26, 28]. For example, on amazon.com, a delay of 100ms costs in average 1% of sales [26]. In order to deliver fast response and offline support, a number of web applications started caching data on the edge. Facebook designed News Feed [2] to support offline access; Google Docs and Google Maps can also be used offline via Chrome browser extension [3].

Many prior work efforts have studied data management in settings where clients are intermittently connected to servers or to peers. Bayou [42] pushed data replicas to the edge in the context of mobile environments (Terry [41] presents an excellent synthesis on the topic), then Cimbiosys [36] extended the decentralized synchronization model to Internet Services, in addition to Rover [23] and Coda [25], those systems supports disconnected operations but rely on a weak consistency model.

Recently, Parse [14] and Cloud Types [15] are programming models for shared cloud data, they allow local data copies to be stored on the edge client and later be synced with the cloud, but provides only an eventual consistency model.

In prior work, we have explored protocols for partial replication on clients extending a geo-replicated datastore. Swiftcloud [47] allows programmers to dynamically specify a set of objects that is replicated on clients residing in points-of-presence. It allows an offline-first approach with low latency by building on CRDTs and transactional causal consistency. Swiftcloud targets the same high availability techniques as PRACTI [12] and Depot [33], but the later two uses a fat metadata approach (version vectors sized as the number of clients) and they support only LWW registers (but their rich metadata design could support CRDTs too). Swiftcloud further guarantees that updates are neither duplicated nor lost when failing over to other DCs in case of (temporary or final) disconnection with some DC. Depot [33] support Byzantine faults tolerance, a more difficult class of faults than Swiftcloud. However it is not designed to scale to large numbers of clients, to co-locate data with the user without placing a server in the edge machine, nor does it support transactions. Recently, Simba [35] provides the ability for the edge application to select the level of desired observable consistency (eventual, causal or serializability).

In our current work on EdgeAnt (3.1), we retarget our work on Swiftcloud to AntidoteDB with extensions to direct communication between edge clients. To improve edge-to-edge latency, and give collaborative applications the ability to select stronger (than causal) consistency guarantees.

More and more applications supports collaboration between edge devices by storing partial shared data in client machines. From medical editors like [10] that uses Google Drive Realtime [13] a framework that manages shared replicas between clients in the same way as Mobius [17] that is used in mobile context, to FPS multiplayer games and collaborative document editing [24] where a subset of client needs high availability and serializability on shared metadata. Unlike EdgeAnt, those systems rely on an always-up connection to centralized server. On the other end, iCloud [9] provides network independence, internet disconnected devices can share data (files, calls or text messages) via bluetooth. EdgeAnt also provides topology independence, with dynamic reconfiguration of peer-to-peer edge groups.

Offline support for mobile/web apps In the context of web and mobile app developments, there is a number of established frameworks and libraries for partial replication of data on client devices. These frameworks support programmers in developing Progressive Web Applications (PWAs)[7] which support - beside other features - offline operation. As one major component in such an app stack, Service workers[5] provide mechanisms for background synchronization, rerouting of requests and receiving of updates in cross-platform PWAs.

PouchDB[6] is an open-source JavaScript database that eases the synchronization with CouchDB-compatible servers. To this end, stored documents are extended with meta-data containing the unique, user-defined id and a revision number, which is obtained by hashing the document's content. When synchronizing a new object state with the server database, only updates referring to the current revision number are accepted. Further, concurrent updates on different server replicas can lead to a divergent state, further complicating the app development. Our solution, WebCure 3.1, provides a CRDT-based data model where divergent state can never appear as updates are merged while respecting the causal relation between updates.

Realm Mobile[8] is a framework, which integrates a client-side database for iOS and Android with a server-side database. Realm's approach to conflict handling can be summarized as follows:

- **Deletes always win.** If one side deletes an object, it will always stay deleted, even if the other side has made changes to it later on.
- **Last update wins.** If two sides update the same property, the value will end up as the last updated.
- **Inserts in lists are ordered by time.** If two items are inserted at the same position, the item that was inserted first will end up before the other item. This means that if both sides append items to the end of a list they will end up in order of insertion time.

The authors of the framework state that these rules provide "strong eventual consistency". However, programmers need to incorporate these semantics into their development scheme, whereas WebCure allows for data-type semantics that offer more flexibility.

For a detailed market study on frameworks supporting offline operation for mobile / web apps via partial replication, we refer to D8.3. / D8.6.

Access control A detailed discussion on state-of-the-art for access control in replicated weakly consistent datastore can be found in D6.1.

AQL Geo-replication has become a key feature in cloud storage systems, with data being replicated in multiple data centers spread around the world. The goal of geo-replication is to provide high availability and low latency, by allowing clients to access any nearby replica. To achieve these properties, a number of systems [19, 32] adopt a weak consistency model, where an update can execute in any replica, being propagated asynchronously to other replicas.

Writing correct applications under weak consistency can be complex. To address this problem, several geo-replicated storage systems [4, 18, 44] adopt a strong consistency approach. While several optimization techniques have been proposed for improving throughput [18] and latency [34], executing operations involves inter-data-center coordination, with impact on latency and availability.

AQL is closer to systems [30, 39] that provide support for both weak and strong consistency. For helping programmers decide which operation should execute under each consistency model, several tools have been proposed [11, 22, 29, 30, 37]. These tools, typically based on static analyses, impose an additional complexity to application development that is often non-trivial and require analyzing the application code before execution. In AQL, the programmer specifies the degree of concurrency allowed and which database constraints should be maintained – the system enforces the specified concurrency in runtime while trying to minimize coordination. Unlike previous approaches, AQL does not require a prior analysis of the application code, thus supporting access from multiple applications and application evolution.

AQL provides a consistency level that extends parallel-snapshot isolation [40] with integrity invariants, in a similar way as snapshot isolation has been extended with integrity invariants [31]. Our approach for enforcing referential integrity under weak consistency can be seen as an extension of the approach to enforce serializability under snapshot isolation proposed by Cahill et. al. [16], be executing additional updates to force concurrency detection, and using conflict resolution policies to achieve the intended behavior.

Multi-cloud query federation Multi-cloud storage is an emerging data storage scheme used by various types applications. The goal of multi-cloud storage is to enable applications to use multiple public, and optionally private, cloud storage services transparently, through a single interface. To achieve this, multi-cloud systems abstract the use of multiple data storage systems, and offer applications a single, unified namespace. Scality's multi-cloud framework, Zenko, enables applications to transparently store and access data on multiple public and private cloud storage systems using a single storage interface (AWS S3 API).

An important aspect of multi-cloud storage is the ability to search and retrieve data across multiple clouds. Zenko provides a federated metadata search functionality, enabling data to be retrieved based on their metadata attribute values independent of the data location. This functionality is the focus of Scality's use case.

There are a number of commercially available tools that can be used off-the-shelf to support federated multi-cloud querying. One approach is to use Apache Spark, and specifically Spark SQL [12]. Spark SQL is a parallel SQL engine built on top of Apache Spark that provides integration between relational and procedural processing. It offers

[12] https://spark.apache.org/sql/

a declarative API that integrates relational operators with procedural Spark code. Object metadata attributes in Zenko's data model can be represented as relational data in a straightforward way. Then metadata queries be implemented using Spark SQL queries. A disadvantage of this approach is that while Spark SQL can query data from external sources, it requires relational data to be imported in a specific semi-structured format (Parquet files). This leads to the need to post-process the contents of Zenko's metadata database to generate parquet files, and keep those files updated. Scality has evaluated the performance of Apache Spark as a tool to perform metadata search. The evaluation [13] showed that Spark incurs a high latency to metadata search operations due to the need to reload all parquet files for every update needs. Also Spark requires a large amount of resources to function properly in this scenario.

Zenko's implementation captures the metadata attributes of each object, along with the object's location and stores them in a metadata database that is part of Zenko's architecture. Another approach to provide multi-cloud metadata search in this setting is to use an existing database with indexing and query capabilities as Zenko's metadata database, and leverage its querying functionality to implement metadata search. There are mutiple data stores that can be used as this metadata database, including Apache Cassandra [14], MongoDB [15], FoundationDB [16], CockroachDB [17] and Tokyo Cabinet [18]. Scality has investigated a number of these options, with the additional requirement that Zenko's metadata database should replicate data within a data centre for fault-tolerance. The current implementation of Zenko uses MongoDB as it offers acceptable performance and is well known in the industry. The inherent drawback of this approach in a multi-cloud environment is that metadata from multiple private and public cloud storage services need to be propagated to the data centre where Zenko is deployed. This leads to high network usage, and potentially stale search results as metadata attributes need to be propagated to Zenko's metadata database and indexed before becoming searchable.

Moreover, various query federation approaches have been proposed in the academic literature in the context of query federation over linked data, and query processing in multistore systems.

Semantic query federation is an approach to query linked data from multiple distributed datasets. Federated query engines [21, 38] for linked data can provide interesting insights for aspects of the problem including data source selection and subquery building. However, these approaches are not directly applicable to the problem of query federation on cloud storage services due to the differences in the data model and types of queries between linked data and object storage.

Multistore [20, 27] systems, provide integrated or transparent access to a number of cloud data stores (NoSQL, HDFS, RDBMS, etc.) through one or more query languages. These approaches focus on the problem of providing integrated access to heterogeneous data by defining a global schema for the multidatabase over the existing data and mappings between the global schema and the local data source schemas. These techniques commonly translate a given query to subqueries which are then executed by the local data

[13] https://www.zenko.io/blog/benchmark-metadata-search/
[14] http://cassandra.apache.org/
[15] https://www.mongodb.com/
[16] https://apple.github.io/foundationdb/
[17] https://www.cockroachlabs.com/
[18] https://fallabs.com/tokyocabinet/

stores. However, in our scenario cloud storage services are less heterogenous and they typically do not provide support for queries on metadata attributes.

Our solution, Proteus, provides multi-cloud querying by maintaining a geodistributed index. It can take advantage of storage systems with querying functionalities or extend systems that do not support querying by building external indexes. Proteus also offers flexible index and computation placement across a geo-distributed multi-cloud system, and flexible query processing by allowing caching and database scanning in addition to indexing, enabling the querying system to make trade-offs according to requirements of each multi-cloud use case.

7 Exploratory work

In addition to the core results described in the previous sections, we investigated on problems directly related to the heavy-edge scenario of WP6, but not reflected in the Lightkone Reference Architecture and the related software artifacts.

(a) Partial replication on the servers

We have been researching solutions for supporting partial replication that can be integrated in AntidoteDB. The key challenge in supporting partial replication in AntidoteDB is related with being able to provide the consistency level of AntidoteDB efficiently. AntidoteDB provides transactional causal consistency, combining causal consistency with highly-available transactions, where a transaction reads from a causally consistent snapshot. Supporting such consistency level in a partially replicated data stores involves devising an efficient way for accessing both causally consistent data and a database snapshot that may include data stores in different data centers.

In this period, as discussed in D3.2, we have devised an algorithm to efficiently access causally consistent data. As our first solution did not support transactions, we have studied the possibility of integrating it with AntidoteDB. We found out that it would require important changes to the core of AntidoteDB's transactional protocol. As such, we have decided to study how to integrate it in Cassandra, a popular geo-replicated data store. We expect to use the lessons learnt to support partial geo-replication in AntidoteDB at a later point.

C^3 We next explain how we have integrated C^3 in Cassandra, to enforce causal consistency in Cassandra. Our design handles three types of operations: *(i) read* operations, to read the state of the database; *(ii) write* operations, to modify the state of the database; and *(iii) migrate* operation, to change the home data center of a client.

Cassandra operations are classified as read or write operations, and their execution follows the proposed algorithm, which is detailed in D3.1 and in the submitted paper listed in the end of this deliverable. The key idea, is to decouple the propagation of causality tracking information and the propagation of operations to remote site. The causality tracking information for an operation, called label, is generated when the client submits an operation. For executing a remote operation, the operation and its label must be received and the operation can only be executed after the dependencies are satisfied locally.

At each moment, our protocol considers that a client has a home data center. A client can only issue read and write operations on data objects that are replicated in its home data center. Thus, prior to execute a read (or write) operation for an object that is not replicated in its home data center, the client must execute a migrate operation. We note that applications do not have to explicitly issue migration operations, as our client layer automatically issues the necessary migrate operations.

To integrate our algorithm with Cassandra, our design needs to: *(i)* guarantee that the storage system conforms the requirements of our protocol; and *(ii)* modify the execution of operations to integrate our replication scheme.

To guarantee that Cassandra conforms the requirements of our protocol, we only need to guarantee that after a write completes in a data center, all following reads observe a database version that reflects the completed write. In AntidoteDB, this is guaranteed, but the same may not be the case in Cassandra. To achieve this property in Cassandra, we need to execute all operations with the *LOCAL_QUORUM* consistency level. After a write completes in a local write quorum, by the properties of the local quorums, it is guaranteed that any local (read) quorum intercepts the local write quorum. Thus, any read will return the value of the write (assuming that no more recent write exists).

To integrate our replication scheme, we needed to make some changes to both the write execution flow and the read repair replication mechanism.

In Cassandra, the execution of a write starts with the client sending the operation to the to-be coordinator node. The coordinator propagates the operation to the relevant nodes in its local data center and to *one* node in each remote data center, which is responsible to forward the operation to the relevant nodes in the data center. The coordinator waits for the confirmation that the operation has been executed in a subset of nodes before responding to the client – the subset of nodes necessary for returning to the client depends on the client's consistency level.

We changed the write execution flow as follows. First, we changed the coordinator code to send the information about the write to the causality layer, as soon as it propagates the operation to the relevant nodes. Second, we changed the code that processes a write received from the coordinator. Instead of executing the write operation immediately, nodes need to wait until both the operation and the label have been received. This was implemented by creating a *message sink* to which all labels and update operations are redirected. An operation is only executed after the label and updated operations are received. Finally, we changed the code that replies to the coordinator after a write operation finishes, by adding code to send an *acknowledge* to the causality layer.

Cassandra also employs a mechanism called *read repair*. This mechanism is used to update replicas that might have not been updated in prior writes (as messages are lost and the coordinator only waits for replies from a quorum of nodes). Since we do not want this read repair mechanism to violate causality, by being executed across data centers and applying updates to nodes where that update's label has not yet been delivered, we modified its behavior in order to execute only inside data centers.

We are currently studying how to extend the proposed approach to support transactions, which is necessary for AntidoteDB. The key problem is that, when accessing a snapshot, it might be necessary to read an object that is not replicated locally. At this moment, it is necessary to contact a remote replica and wait until the transaction snapshot is ready. With the current AntidoteDB protocol, this requires the periodic stability protocol to execute. We are studying if we could build on the migrate operation proposed on C^3 algorithm for speeding up this process.

(b) PoR Consistency

The Just-Right Consistency (JRC) approach introduced in D4.1 aims to minimize the amount of synchronization required to ensure the applications invariants. As mentioned in the previous section, JRC can be built upon common invariant-preserving programming patterns, such as "ordered updates", "atomic grouping" and "precondition check".

We have been exploring different alternatives to implement JRC. In the past, we have proposed RedBlue consistency [30], which allows some operations to execute under strong consistency (and therefore incur a high performance penalty) while other operations can execute under weaker consistency (namely causal consistency). The core of this solution is a labeling methodology for guiding the programmer to assign consistency levels to operations. The labeling process works as follows: operations that either do not commute w.r.t. all others or potentially violate invariants must be strongly consistent, while the remaining ones can be weakly consistent.

This binary classification methodology is effective for many applications, thus implementing the JRC approach, but it can also lead to unnecessary coordination in some cases. In particular, there are cases where it is important to synchronize the execution of two specific operations, but those operations do not need to be synchronized with any other operation in the system (and this synchronization would happen across all strongly consistent operations in the previous scheme). In the past, we have proposed solutions that allow finer-grained coordination of operations [11, 22].

Concurrently with those works, we have also worked on an alternative generic approach, Partial Order-Restrictions Consistency (or short, *PoR* consistency), which takes a set of restrictions as input and forces these restrictions to be met in all partial orders. This creates the opportunity for defining many consistency guarantees within a single replication framework by expressing consistency levels in terms of visibility restrictions on pairs of operations. Weakening or strengthening the consistency semantics is achieved by imposing fewer or more restrictions, thus implementing the JRC general idea.

Under *PoR* consistency, the key to making a geo-replicated deployment of a given application perform well is to identify a set of restrictions over pairs of its operations so that state convergence and invariant preservation are ensured if these restrictions are enforced throughout all executions of the system. To this end, we propose a set of principles for guiding programmers to identify the important restrictions while avoiding unnecessary ones.

Furthermore, from a protocol implementation perspective, given a set of restrictions over pairs of operations, there exist several coordination protocols that can be used for enforcing a given restriction, such as Paxos, distributed locking, or escrow techniques. Depending on the frequency over time with which the system receives operations confined by a restriction, different coordination approaches lead to different performance trade-offs. Therefore, to minimize the runtime coordination overhead, we also developed an efficient coordination service that helps replicated services use the most efficient protocol by taking into account the system workload.

When compared with other works that also allow finer-grained coordination of operations [11, 22], this work makes the following main contributions. First, we propose a method to find a minimal set of restrictions to be used. Second, we designed a set of coordination methods that can be used with different workloads, with applications being modified directly from the identified set of restrictions. More information is available in our ATC'18 paper, presented in the end of this report.

8 Publications and Dissemination

The following publications and dissemination activities have been done under WP6 in the report period M13 - M18.

8.1 Refereed conference and workshop papers

- Cheng Li, Nuno Preguiça, Rodrigo Rodrigues. Fine-grained consistency for geo-replicated systems. USENIX Annual Technical Conference 2018: 359-372
- Pedro Fouto, Joo Leitão, Nuno Preguiça. Practical and Fast Causal Consistent Partial Geo-Replication. NCA 2018: 1-10
- Zhongmiao Li, Peter Van Roy, Paolo Romano. Transparent Speculation in Geo-Replicated Transactional Data Stores. HPDC'18, June 11 - 15, 2018. (Description of this work was given in D6.1)
- Dimitrios Vasilas, Marc Shapiro and Bradley King. A Modular Design for Geo-Distributed Querying: Work in Progress Report, Int. Workshop on Principles and Practice of Consistency for Distributed Systems (PaPoC 2018), Porto, Portugal, April 23 - 26, 2018.

8.2 Under submission

The following works are under submission:

- Zhongmiao Li, Peter Van Roy, Paolo Romano. Sparkle: Fast and Scalable Deterministic Partitioned Datastore. Under submission.

8.3 Theses

- Angel Manuel Bravo Gestoso. Metadata Management in Causally Consistent Systems. Doctoral Thesis, Universidade de Lisboa, Instituto Superior Tecnico, and Universite Catholique de Louvain, June 2018.
  https://www.info.ucl.ac.be/~pvr/manuelbravo-thesis.pdf

8.4 Talks

- Annette Bieniusa. *AntidoteDB*. PL4DS: Workshop on Programming languages for Distributed Systems. Feb. 23, 2018, Darmstadt (Germany).
- Nuno Preguiça. *Enforcing SQL constraints in weakly consistent databases*. Presentation at Dagstuhl Seminar 18091, Feb. 25 - Mar. 2, 2018, Wadern, Germany.
- Nuno Preguiça. *Getting stronger with AntidoteDB*. Invited talk at Lisbon IPFS Hack Week, May 2018.
- Annette Bieniusa. *Just the right kind of Consistency!*. Keynote talk at Typelevel Summit Berlin, May 18, 2018, Berlin, Germany. See https://typelevel.org/event/2018-05-summit-berlin/.

- Marc Shapiro. *Antidote: A developer-friendly cloud database for Just-Right Consistency*. Keynote talk at 4th International Conference on Advances in Computing & Communication Engineering (ICACCE 2018), June 2018, Paris, France.

References

[1] Akamai new study reveals the impact of travel site performance on consumers. http://www.akamai.com/html/about/press/releases/2010/press_061410.html. Published: June 14, 2010.

[2] Facebook: continuing to build news feed for all types of connections. https://code.fb.com/android/continuing-to-build-news-feed-for-all-types-of-connections/l. Published: Dec 09, 2015.

[3] Google Docs: offline access. https://support.google.com/docs/answer/6388102?co=GENIE.Platform%3DDesktop&hl=en.

[4] Microsoft CosmosDB. https://docs.microsoft.com/en-us/azure/cosmos-db/. Accessed Ago-2018.

[5] Introduction to service worker. https://developers.google.com/web/ilt/pwa/introduction-to-service-worker, 2018 (accessed November 18, 2018).

[6] Pouchdb - the database that syncs! https://pouchdb.com, (accessed January 10, 2019).

[7] Progressive web applications. https://developers.google.com/web/progressive-web-apps/, (accessed January 10, 2019).

[8] Realm. https://realm.io, (accessed January 10, 2019).

[9] Apple. Apple icloud. https://icloud.com.

[10] Istemi Bahceci, Ugur Dogrusoz, Konnor C La, Özgün Babur, Jianjiong Gao, and Nikolaus Schultz. Pathwaymapper: a collaborative visual web editor for cancer pathways and genomic data. *Bioinformatics*, 33(14):2238–2240, 2017.

[11] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[12] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *NSDI*, volume 6, pages 5–5, 2006.

[13] Google Blog. Google drive reatime api. https://developers.google.com/realtime/overview.

[14] Parse Blog. Parse: website. https://parseplatform.org/.

[15] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. Cloud types for eventual consistency. In *European Conference on Object-Oriented Programming*, pages 283–307. Springer, 2012.

[16] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM.

[17] Byung-Gon Chun, Carlo Curino, Russell Sears, Alexander Shraer, Samuel Madden, and Raghu Ramakrishnan. Mobius: unified messaging and data serving for mobile apps. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 141–154. ACM, 2012.

[18] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed Database. In *Proc. 10th USENIX Conf. on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. 21st ACM SIGOPS Symp. on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[20] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. The bigdawg polystore system. *SIGMOD Rec.*, 44(2):11–16, August 2015.

[21] Olaf Görlitz and Steffen Staab. Splendid: Sparql endpoint federation exploiting void descriptions. In *Proceedings of the Second International Conference on Consuming Linked Data - Volume 782*, COLD'11, pages 13–24, Aachen, Germany, Germany, 2010. CEUR-WS.org.

[22] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM.

[23] Anthony D Joseph, Alan F de Lespinasse, Joshua A Tauber, David K Gifford, and M Frans Kaashoek. Rover: A toolkit for mobile information access. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 156–171. ACM, 1995.

[24] josephg. Sharejs. https://github.com/josephg/ShareJS.

[25] James J Kistler and Mahadev Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.

[26] Ron Kohavi and Roger Longbotham. Online experiments: Lessons learned. *Computer*, 40(9), 2007.

[27] Boyan Kolev, Carlyna Bondiombouy, Patrick Valduriez, Ricardo Jimenez-Peris, Raquel Pau, and José Pereira. The cloudmdsql multistore system. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2113–2116, New York, NY, USA, 2016. ACM.

[28] Tom Leighton. Improving performance on the internet. *Communications of the ACM*, 52(2):44–51, 2009.

[29] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association.

[30] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[31] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño Martínez, and José Enrique Armendáriz-Iñigo. Snapshot Isolation and Integrity Constraints in Replicated Databases. *ACM Trans. Database Syst.*, 34(2):11:1–11:49, July 2009.

[32] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proc. 23d ACM Symp. on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[33] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):12, 2011.

[34] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1221–1236, New York, NY, USA, 2018. ACM.

[35] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V Madhyastha, and Cristian Ungureanu. Simba: Tunable end-to-end data consistency for mobile apps. In *Proceedings of the Tenth European Conference on Computer Systems*, page 7. ACM, 2015.

[36] Venugopalan Ramasubramanian, Thomas L Rodeheffer, Douglas B Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, 2009.

[37] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction

coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1311–1326, 2015.

[38] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, pages 601–616, Berlin, Heidelberg, 2011. Springer-Verlag.

[39] Swaminathan Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[40] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proc. 23d ACM Symp. on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[41] Douglas B Terry. Replicated data management for mobile computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 3(1):1–94, 2008.

[42] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 172–182. ACM, 1995.

[43] Dimitrios Vasilas, Marc Shapiro, and Bradley King. A modular design for geo-distributed querying: Work in progress report. In *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '18, pages 4:1–4:4, New York, NY, USA, 2018. ACM.

[44] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052, New York, NY, USA, 2017. ACM.

[45] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, Berkeley, CA, USA, 2014. USENIX Association.

[46] Marek Zawirski, Carlos Baquero, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. Eventually consistent register revisited. In Peter Alvaro and Alysson Bessani, editors, *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, pages 9:1–9:3. ACM, 2016.

[47] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, pages 75–87. ACM, 2015.

[48] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, New York, NY, USA, 2015. ACM.