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

D4.1: Basic programming model for edge computing

Deliverable no.:	D4.1
Title:	Basic programming model for edge computing
Due date of deliverable:	January 15, 2019
Actual submission date:	January 15, 2019
Lead contributor:	UCL
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
05/12/2018	1.1	File creation	UCL
05/12/2018	1.2	Initial structure	UCL
06/01/2019	1.3	Refined structure and first SOTA	UCL
09/01/2019	1.4	First complete content	UCL
13/01/2019	2.0	First complete deliverable	UCL

Detailed revision information is available in the private repository https://github.com/LightKone/WP4.

Contributors:

Contributor	Institution
Peter Van Roy	UCL
José Proença	INESC TEC
Carlos Baquero	INESC TEC
Marc Shapiro	Sorbonne-Université
João Leitão	NOVA ID

Contents

1 Executive summary

Main progress The main progress of the first year is given by the following items:

- *Basic programming model.* Successful design of a single programming model for edge computing, implemented in the LightKone Reference Architecture (LiRA) which is defined in Deliverable D3.1. The programming model is defined by a formal semantics that subsumes Lasp and Antidote, which are the two starting points for LightKone that each represent an extreme case for edge computing, namely light edge and heavy edge (as defined in the grant agreement). Both Antidote and Lasp are based on CRDTs (Conflict-Free Replicated Data Types), which are replicated distributed data structures that provide consistency in highly efficient manner due to their mathematical structure (no consensus needed between replicas). Other than both using CRDTs, Antidote and Lasp have little in common a priori, so the development of the unified semantics was an important step for LiRA.

- *Just-right consistency (JRC).* Elaboration of a programming methodology for building highly available distributed applications that maintain consistency despite frequent concurrent operations and network partitions. This methodology is considered essential for the development of heavy edge applications.

- *Legion progress.* Legion is a new LiRA component that provides data sharing and communication among Web clients, such as mobile applications. Legion covers an important part of the edge computing application space.

- *Antidote progress.* Antidote is a LiRA component that supports heavy edge applications. Continued development of the Antidote database system, including both implementation and semantics. In particular, Antidote implements the JRC methodology and we have implemented an SQL layer for Antidote, called AQL (Antidote Query Layer).

- *Lasp progress.* Lasp is a LiRA component that supports light edge applications. We have successfully scaled up Lasp to run on networks of 1024 nodes, which required significant engineering effort and the design of a new *workflow CRDT*,

Lasp, Antidote, and Legion are released as software components to third parties.

Main innovations with respect to state of the art We compare LiRA's programming model progress with four major existing architectures for edge computing, namely the OpenFog Reference Architecture, Microsoft Azure IoT, Amazon IoT Greengrass, and ECC Edge Computing. With respect to the programming models of these architectures, the main innovation of LiRA is the *convergent data store*, which provides a single API that combines four abilities, namely resilient data storage, resilient communication, distributed consistency, and dynamic network support (node turnover). In the state of the art, the burden of combining these four properties is put on the shoulders of the developer, who must orchestrate interactions between storage, communication, and computation services, in order to satisfy distributed consistency. In LiRA, this burden is handled by the architecture. This innovation is highly relevant because networks on the edge have unreliable nodes and unreliable communication, and are dynamic.

Exploratory work In addition to LiRA innovation, we have also done significant exploratory work in the area of programming models. This exploratory work is an essential part of a European-funded RIA because it provides early investigation of ideas for future advances in the state of the art. Regarding LiRA, we have explored several possible future directions for its evolution:

- Available file system. We investigate extensions to the standard Posix file system semantics targeted toward distributed file systems running on unreliable networks, e.g., subject to network partitioning.
- Quality-aware reactive programming. We investigate how quality-aware information can propagate in the execution of a program, so that the program can react quickly to changes in system reliability (unreliable nodes and communication). This is relevant for maintaining acceptable performance on light edge despite infrastructure unreliability.
- Rethinking distributed programming. The main LightKone innovation, as presented above, is a step in the direction of abstracting away the inessential difficulties of building distributed applications, leaving only the essential difficulties to be managed by the developer. We have investigated further steps in this direction. For example, instead of treating database contents as truth, an edge computing system could invert its structure and treat the edge data (e.g., sensor data) as truth and database contents as an approximation that converges toward the truth. Because convergence can be made automatic, this has the potential to simplify the developer's task.

2 Introduction

The present deliverable documents the progress made on programming models in LightKone during the first period (Month 1 - Month 12). Sufficient information is given in the main body of this document (i.e., without the appendices or scientific papers) to evaluate this progress. The document is structured as follows:

- Progress and plan (Section 3). This section summarizes the project plan regarding programming models, and gives the Month 12 snapshot of the progress with respect to this plan. Progress made with respect to the project milestones is presented. Summaries are given of the major items of progress in programming models.
- Software (Section 4). This section gives the links how to access the software deliverables and their documentation.
- State of the art (Section 5). This section summarizes the state of the art in programming models for edge computing, and gives the innovation provided by LightKone with respect to this state of the art.
- Exploratory work (Section 6). This section gives the exploratory work performed, i.e., the research-oriented work on programming models that is important for the future evolution of LiRA.
- Published papers (Section 7). This section lists papers published by LightKone that support this deliverable. The content of the papers is available on the LightKone web site.
- Other dissemination (Section 8). This section lists other dissemination activities related to this work package, in particular invited talks and submitted papers by LightKone that support this deliverable.
- Unified semantics for LiRA (Antidote and Lasp) (Appendix A). This appendix complements the programming model progress summary of Section 3. It is included as reference because it has not been published elsewhere.
- Operational semantics for Antidote (Appendix B). This appendix complements the programming model progress summary of Section 3. It is included as reference because it has not been published elsewhere.
- References (Appendix C). This section gives bibliographic references to published articles outside of the project (either by partners before the project, or by third parties) that support this deliverable's work.

All the work explained in this deliverable is supported by released and documented software and by other documents including published scientific papers. We recall that it is not necessary to read the scientific papers nor the appendices to evaluate the progress of the programming model work; all necessary information is given in the main text of this deliverable. They can be used as references, to clarify any point or to give extra information where necessary.

2.1 Summary of deliverable revision

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

- The LightKone Reference Architecture (LiRA) is defined in Deliverable D3.1, as requested by the Reviewers, and the programming model related part of LiRA is explained in the present document.
- Explanation of the need to define a new semantics is given in Section 3.3 and Section 5. As explained in these sections, the LiRA programming model itself is an innovation that does not exist in any of the major edge computing architectures. In brief, the new semantics exists to ensure that developers using LiRA see no unpleasant surprises and to pave the way for coherent future enhancements of LiRA. The new semantics allows us to reduce risks when evolving LiRA.
- The achieved results of work packate 4 during the first period are fully described in the present document and not redirected to scientific papers. Note that this was already the case in the original submission of Deliverable D4.1, where the scientific papers were included only as reference but not needed for evaluation. The present document does not include the scientific papers; they are all available on the project web site.

3 Progress and plan

We present the progress made on the programming model for the LightKone reference architecture during the first 12 months. Comparison with state of the art and explanation of innovation is done in Section 5.

3.1 Plan

We present the plan for programming models for edge computing, year by year. Note that for convenience this plan is divided into three phases, one per year, but the actual work of the phases will of course overlap as needed.

- **Year 1**. The first year of the project defined a basic programming model for edge computing, which is supported by a set of components in the reference architecture. The purposes of this first model is to show feasibility of general-purpose edge computing with respect to the basic functionality that we estimated will be needed by the use cases. This programming model is supported by three components in the reference architecture, namely Antidote, Legion, and Lasp. Antidote was originally designed for cloud environments, and it is being extended for the edge. Legion is designed for mobile applications, and runs directly on client nodes (e.g., mobile phones). Lasp was designed for running on dynamic networks, such as edge networks, and will be ported on edge networks in the second year of the project. Lasp uses a fourth component, Partisan, which provides for resilient communication on dynamic edge networks.

- **Year 2**. During the second year of the project, the programming model will be extended as needed for relevant scenarios from the use case scenarios in WP 2. The second model will extend the first model with relevant functional and non-functional properties. Specifically, Lasp will be ported on the GRiSP embedded systems board and extended to allow it to perform computations directly at the edge by adding a task model. This will allow applications to run completely on the GRiSP board, completely at the edge without necessarily requiring connections to a gateway or to cloud for normal functioning. Basically, these connections will be used to *manage* the application, but not for moment-by-moment *running* of the application. Furthermore, Antidote will be extended with an edge client, called EdgeAnt, that allows it to support heavy edge applications.

- **Year 3.** In the third and final part of the project, we will focus on the implementation and evaluation of use case scenarios chosen from the WP 2 industrial partner use cases. The programming model will then be targeted toward these use cases and will be improved for scalability and robustness through feedback from the evaluations of WP 7.

3.2 Progress

In this subsection we summarize the progress in programming models during the first year of the project. These programming models are supported by LiRA, as presented in Deliverable D3.1. Section 5 compares this work with the state of the art and gives the innovations that it represents.

Regarding project milestones during the first period, work package 4 delivered MS2. This milestone marks that the generic edge model is ready for application development. With respect to the programming model, this milestone has been achieved through the design of the unified semantics. With respect to the implementation, the progress is explained in Deliverable D3.1.

(a) Basic programming model

The main result of the first year, in terms of programming models, is the successful design of a unified programming model for edge computing. This programming model underlies LiRA and is supported by the main components in LiRA, as they are presented in Deliverable D3.1. Because of this programming model, LiRA is more than just a collection of components; rather, the components work together to provide a simple model for application developers. The programming model provides a single API that is provided in three variants (Antidote, Lasp, and Legion), targeted toward three different edge scenarios. Section 3.3 gives more information on this contribution.

(b) Just-right consistency (JRC)

Just-right consistency is a programming methodology for building highly-available distributed applications that maintain consistency despite concurrent operations and network partitions. JRC is being realized for the Antidote platform, with new tools, tutorials, and techniques to facilitate its use by developers. We consider that JRC is important for edge computing because of the dynamicity and unreliability of edge networks. The effectiveness of JRC will be evaluated during the last year of the project. Section 3.4 gives more information on this contribution.

(c) Legion progress

Legion is a LiRA component that provides data sharing and communication among Web clients, such as mobile applications. Legion uses CRDT data structures and peer-to-peer communication between devices. Because of CRDT semantics, Legion provides data consistency requiring only eventual peer-to-peer communication between devices. Legion is able to use centralized services if necessary for data durability. Legion provides a library of useful CRDT data types for the application developer. Work is being done on improving scalability of Legion applications, to allow hundreds or more of clients. Section 3.5 gives more information on this contribution.

(d) Antidote progress

Antidote is a LiRA component that provides transactional database abilities with causal consistency and support for heavy edge applications (a.k.a. fog computing). These abilities go significantly beyond the state of the art: in particular since Antidote uses CRDT data types, it is able to commit transactions even if there are network partitions, without sacrificing consistency (see Section 5 for more explanation). The Antidote software system is in continuous development toward making it ready for building real applications; the progress in this area is presented in Deliverable D6.1. In the present deliverable, we focus on the programming model aspect of Antidote. In this area, progress was made in two areas. First, the definition of an operational semantics for Antidote that ensures that the API provides a solid behavior with no unpleasant surprises. Second, the addition of an SQL layer for Antidote, called AQL (Antidote Query Layer).

(e) Lasp progress

Lasp is a LiRA component that provides the ability to do reliable computation directly on edge devices, given a limited amount of computing power and communication ability. Lasp uses CRDT data structures and the Partisan library for reliable communication. Lasp goes significantly beyond the state of the art in providing reliable computation directly on edge devices. No gateway or Internet connection is necessary for Lasp operation (although of course they can be used, e.g., for periodic system management), which we call light edge applications. Progress on Lasp implementation is presented in Deliverable D5.1. In the present deliverable, we focus on the programming model aspect of Lasp. In this area, we have successfully scaled up Lasp to 1024 nodes (as presented in PPDP 2017, see Section 7.1). Significant engineering effort is required to successfully build and measure successfully measure systems at this scale. In our case, this required adding new programming abilities for deployment and orchestration, as encapsulated in the workflow CRDT.

3.3 Basic programming model

A programming model is exposed to a developer as an API, and provides to the developer a set of operations that satisfy a particular semantics. The APIs of the relevant LiRA components are presented in the software deliverables (see Section 4). In this section we explain the semantics underlying these APIs and how this is applied to Lasp and Antidote. The integration of Legion into the semantics is the subject of future work.

The full semantics of the programming model is presented in Appendix A, followed by the specific semantics of Antidote operations in B. Neither of these documents have yet been published in a scientific paper, and for this reason and because of their brevity, we include them in this deliverable. However, we emphasize that it is not necessary to read them in detail; they are provided only as a reference. All relevant information for evaluating the project is provided in the main body of the deliverable.

LiRA, the LightKone Reference Architecture defined in Deliverable D3.1, supports the semantics defined in the present deliverable (and its further development in the project). The semantics is exposed to programmers through the relevant LiRA components. In this way, both Antidote and Lasp are a coherent part of LiRA. In the future, extensions of LiRa components will be done as required by the implementation and evaluation of the use case scenarios, but always in accord with the unified semantics. The existence of this semantics is very important, even if developers never see it directly (they only interact with APIs): the existence of the semantics guarantees that LiRA components will behave well with no unpleasant surprises.

(a) Motivation and approach

The LightKone project started with two independent platforms for synchronization-free programming, namely Lasp and Antidote, which were both developed in the previous project SyncFree. Both platforms are based on CRDTs, but they developed in opposite directions: Lasp provides dataflow computation at the edge and Antidote provides a transactional database in data centers. They explored complementary directions of how to use CRDTs, which are distributed data structures that provide consistency even with very weak synchronization.

Lasp and Antidote are two separate implementations with significant development effort put into each. It was clear from the beginning that we needed to merge these implementations into a single reference architecture:

- All applications would be able to take advantage of the functionality of both platforms.
- Applications could be distributed both on the edge and in data centers.
- New functionalities that take advantage of both platforms could be introduced, such as edge transactions and data center dataflow.
- Project resources would be better used, since there would be no duplicate implementation efforts.

However, there were two obstacles to overcome:

- Since each of the two platforms already has significant invested resources, making a single reference architecture would seemingly require a huge reimplementation effort.

- For the reference architecture to be useful, we would have to answer the conceptual question of how the Lasp and Antidote functionalities can be combined.

At the LightKone Kickoff meeting in January 2017, the work on the reference architecture started. The first step was to understand conceptually what it means to merge the Lasp and Antidote approaches. We started by defining an operational semantics for Lasp and for Antidote. The Lasp operational semantics is defined in [5]; the Antidote operational semantics is defined in Appendix B. We merged these separate lines of work using event visibility semantics, as defined by Sebastien Burckhardt in his book on eventual consistency [4]. Once this approach was decided on, progress was rapid and led to the semantics defined in the present report. The visible result of this work is the coherence of the LiRA architecture: the components work together and the API variants (targeted toward different edge scenarios) are doing the same data structure operations.

(b) Programming model semantics

This section gives a summary of the programming model semantics. The full semantics is given in Appendix A, which is provided as a reference in this deliverable since it has not been published yet. The semantics defines systems in terms of their abstract executions, where an *abstract execution* is a set of events connected by their visibility. An *event* is an interaction between a client and the system. Each event can see some past events, which is defined by the *visibility relation*. Formally, an abstract execution is a directed graph where the nodes are the events and the edges are the visibility relation between events. We explain the intuition of these ideas below. Further development of LiRA commits to respect this semantics. Exploration of new areas of edge computing will result in extensions of the semantics.

Events and visibility The two basic parts of an abstract execution are events and their visibility.

- An event e defines an operation on an object. Events contain several pieces of information: a key, an operation, and a result value. The set E denotes the set of all possible events, so that $e \in E$. Each event has a *key* k, where $k = \mathrm{key}(e) \in \mathit{Keys}$, where k uniquely identifies the object. The operation $\mathrm{op}(e) \in \mathit{Ops}$ performed on the object. The result value of the operation $\mathrm{res}(e) \in V$ where V is the set of possible result values.

- The visibility relation $\mathit{vis} \subset E \times E$ that defines for each event what other events are visible to it.

Correctness An abstract execution is *correct* if it satisfies the following conditions:

- Acyclic visibility: there are no cycles in the *vis* relation.

- Total arbitration: it is possible to order all events. This is important to avoid nondeterminism in the semantics. An arbitration relation *ar* is added so that concurrent events can be ordered.
- Per-object eventual consistency: all of an object's events are seen by all but a finite number of the object's other events.
- Correct data types: The $\text{res}(e)$ function gives the same result as $F_T(c)$ where c is the context of event e.
- Causality: the *vis* relation is transitive.

Data types Each object in an abstract execution has a type T. The value of an object is defined at each event e. The value does not depend on e by itself, but on all the events that are visible to e, which is known as e's *context*. The data type is defined by a function F_T that takes a context and returns the value. Both Antidote and Lasp support a number of data types, each with their particular function F_T.

Lasp semantics Lasp semantics are defined by adding two concepts to the basic abstract executions, namely *Lasp objects* and *links*. First we partition the key space *Keys* into base objects and Lasp objects. Then we link each Lasp object to n other objects (either Lasp objects or base objects). The link is defined by a list of object keys and a function: $([k_1,...,k_n],f)$. The function defines how the value of the Lasp object depends on the n objects it depends on. Following the semantics of [5], the Lasp implementation supports a number of link operations, namely map, product, intersection, filter, and fold.

Lasp allows linking data items by operations, so that if the input is updated, the output is computed automatically. This follows the convergence of CRDTs: both input and output will converge. We define the *convergence* between linked objects as follows. If a Lasp object k_a depends on an object k_b, then there is eventual consistency between the two objects: all events of k_b are seen by all but a finite number of k_a's events. This assumes there are an infinite number of events for k_b; if this is not true (since Lasp objects are not necessarily read infinitely often), then a slightly different definition is needed (as explained in Appendix A).

Antidote semantics Antidote semantics are defined by adding two concepts to abstract executions, namely *transactions* and *versioning*:

- Each event is associated with a transaction identifier t. We define *atomic visibility* between two transactions t_1 and t_2 so that all the events of t_1 and all events of t_2 are either mutually visible in the same order, or mutually invisible.
- Each event is also associated with a version, which is a set of events. We can define snapshots according to how versions affect visibility. Min snapshot is defined so that if an event is in another event's version, then the event is also visible to that event.

Antidote provides both atomic visibility and min snapshots. Since the abstract execution does not model time, we assume that all transactions are committed. We also do not currently define isolation levels. Both extensions are the subject of future work.

Correctness of the implementation The semantics can be used to prove correctness of the implementation. The basic abstract execution, as introduced above, models the relevant events of an execution without defining their distributed behavior (i.e., on which nodes they execute). The execution is called *abstract* because it does not show on which nodes the events are executed; a *concrete* execution does show the node of each event. We extend the abstract semantics by adding the distributed behavior, namely nodes, node states, and messages between nodes. This gives the concrete semantics, which directly corresponds to the execution of the distributed algorithms in the implementation. Given a concrete execution, we consider the interaction between clients and the system, which is called an observable history. If an observable history can be extended to a valid abstract execution, then the concrete execution is correct. With this approach, we can prove that the Lasp and Antidote protocols satisfy the unified semantics.

3.4 Just-right consistency (JRC)

Just-Right Consistency is a programming methodology to write highly-available, CRDT-based applications that maintain their invariants despite concurrency and occasional network partitioning [6]. Future work will explore how to push just-right consistency toward the edge. This gives a methodology that is closely related to the vision of decentralized truth presented in Section (b).

(a) JRC in a nutshell

When designing a distributed application, the choice of consistency model directly impacts safety and performance. No single model is universally best: synchronous models are safest, but asynchronous ones are fast and tolerate partition. Our Just-Right Consistency (JRC) approach aims to minimise the amount of synchronisation, while ensuring that the application's invariants are safe. It is a provably correct way of tailoring consistency to specific application requirements.

JRC builds upon common invariant-preserving programming patterns. Two patterns (ordered updates and atomic grouping) are compatible with concurrent and asynchronous updates. Another (precondition check) is sensitive to the CAP impossibility [7], but requires synchronisation only in certain cases (unstable precondition).

JRC is practical. Concurrent updates are supported by the CRDT data model. The Transactional Causal Consistency model supports the first two patterns. Encapsulated data types, such as the Bounded Counter, support common cases of CAP-sensitive patterns efficiently. Our CISE and Repliss static analysis tools (both presented in Deliverable D6.1) proves when preconditions are stable and CAP-sensitive updates can safely execute concurrently.

(b) Starting point: a safe sequential application

To be concrete, consider the FMKe application, described in Deliverable D8.2. FMKe simulates a healthcare network and manages the data and events relating patients, doctors, hospitals and pharmacies. It must maintain invariants, for instance that a doctor signs the prescription before the pharmacy will process it, and the pharmacy does not deliver more medication than prescribed. To maintain these invariants, a sequential version of FMKe will follow some typical patterns. The first is to perform operations in a certain order, which is significant. The second is to make several database accesses in a same operation. Finally, preconditions: e.g., the pharmacy checks that a medication has not already been delivered. These three patterns —mutual ordering, joint access, and precondition-update— are critical to safety. We do not ask the developer to make her invariants explicit, but we assume that operations are correct, i.e., when the database starts in a safe state, any application-level operation run in isolation leaves the database in a safe state. This is the Correct-Individually property, the C of ACID.

JRC is a methodology for moving such a sequential operation to a geo-distributed, highly available environment. We assume that the application represents its data as CRDTs in order to allow concurrent updates. However, a major challeng remains: to ensure that invariants remain satisfied. We discuss our approach in the next few sections.

(c) Leveraging the order of operations is AP-compatible

The first pattern is ordering updates. For instance, the doctor signs a prescription before sending it to the pharmacy. If events are delivered in the wrong order, an incorrect state can be observed, e.g., the pharmacy sees an unsigned prescription. Therefore, Just-Right Consistency requires to respect the mutual ordering of events.

This is achieved by a data store that guarantees *Causal Consistency* (CC). Under CC, if update v depends on operation u, then every observer sees u before v. CC transparently maintains the correct order from the sequential application, without any effort from the developer. Unrelated (concurrent) updates can become visible in any order.

CC is compatible with AP [2], because the store only might have to buffer v until u has been delivered. The Antidote store developed in LightKone is an AP store that guarantees CC.

(d) Grouping is AP-compatible

Our second pattern concerns a group of database accesses happening jointly. Just-Right Consistency requires to respect this grouping.

FMKe offers several examples. For instance, signRx reads the patient and doctor IDs from a prescription, and updates the corresponding patient, doctor and prescription objects. Intuitively, all the reads of a group should come from a same snapshot, and its writes be visible in the store at once, in an all-or-nothing manner. Otherwise, for instance, there could be a state where a prescription is referenced by the patient object, but not by the pharmacy object, or vice-versa.

Such atomic grouping can be implemented in an asynchronous manner, since taking a snapshot of the database state does not require any synchronisation between remote replicas, nor does applying several updates at once. Our Antidote data store supports all-or-nothing grouping, in addition to causality; we call this model Transactional Causal Consistency (TCC).

(e) CAP-sensitive safety pattern: precondition-update

Our final *precondition-update* is the case where an operation checks a *precondition* before doing some *update*, to ensure that the final state will be safe (assuming the initial state was safe). If every operation does this, this is the C property of ACID. For instance, processRx checks that a medication's count is greater than the quantity to be delivered;

if so, the count decreases by that quantity. Implicitly, the code is maintaining the invariant count ≥ 0.

This is where CAP comes in: under partition, the state might change at a remote replica, falsifying the local precondition check. Although the operation is safe in isolation, asynchronous concurrent updates can break it.

Even for a CAP-sensitive pattern, not all executions need to be synchronised, because not all preconditions are unsafely affected by concurrency. Consider a database state where the precondition of some operation u is satisfied. If, in this state, executing update v will never make the precondition of u false (we say the precondition of u is stable under v), and vice-versa, then u concurrent with v is OK. We synchronise *only when strictly necessary* because of non-stability. Analysing precondition stability is tractable, because you only need to compare operation to operation pairwise (no need to look at all possible combinations) [8].

(f) Helping the developer to apply Just-Right Consistency

To maintain sequential safety in a concurrent setting while maximising availability requires the following: *(i)* Each operation is safe individually; *(ii)* concurrent updates merge and converge; *(iii)* the system obeys the relative order of non-concurrent updates; *(iv)* it enforces groupings; *(v)* when the precondition of some operation u is unstable under operation v, the two operations do not run concurrently.

To help the developer, LightKone is developing supporting tools, briefly listed next.

The Antidote data store described in Deliverable D6.1 is an essential building block for JRC. Indeed, Antidote supports CRDTs efficiently (*(ii)*). It also guarantees Transactional Causal Consistency (*(iii)* and *(iv)*), thus enforcing the two AP-compatible invariant patterns.

Antidote supports concurrency control, required for point *(v)*, through a specific but very common case: the Bounded Counter (BC) replicated data type [3].

The static analysis tools CISE and Repliss serve to prove that operations are safe in isolation *(i)*, that concurrent execution converges *(ii)*, and that concurrent updates are mutually stable *(v)*.

For instance, CISE takes as input a first-order-logic specification of the operations and invariants. It comprises three main analyses. Analysis 1 verifies that each operation is individually safe (item *(i)*). For any safe combination of starting state and arguments (i.e., the state satisfies the invariant, and the operation's precondition is true in this state with the given arguments), it verifies that the final state is safe. If this verification fails, it means that the operation's precondition was too weak; the tool returns a counterexample that allows the developer to diagnose the issue. Thus, the tool can also be used as a design assistant.

The next two analyses verify *pairs* of possibly-concurrent operations; operations that are never concurrent can skip them. For any safe state/argument combination, Analysis 2 verifies that running the pair in either order yields the same safe final state (convergence, item *(ii)*). Analysis 3 verifies that the precondition of one operation is stable under the side-effects of the other (stability, item *(v)*). If either verification fails, the developer needs either to put in some concurrency control, or to weaken the invariant, as discussed earlier.

If all three pass, this constitutes formal proof that the application invariant remains true at all times [8]. For instance, we proved the safety of FMKe.

(g) AntidoteFS

As a case study of this approach, we are developing AntidoteFS,[1] a distributed file system built on Antidote. The main design goal of AntidoteFS is respecting a near-POSIX semantics while still maintaining high availability in the face of network partitions. Therefore, in this case, the application-level invariants are those defined in the POSIX file system specification. We used the CISE verification tool to identify the CAP-sensitive file system operations, i.e. the operations that require coordination in order to maintain the POSIX invariants. The main output of this analysis has been that operations that move folders have to be synchronized in order to preserve the invariants of the tree-like structure of the file system. Therefore, as next step, we intend to judiciously blend different coordination protocols to implement a consistency model that just matches the semantics of the individual file system operations. This is in stark contrast with state-of-the-art file systems, which either give up POSIX semantics tout court in favor of availability, or adopt coordination protocols which penalize performance and scalability. Further details on AntidoteFS are reported in D6.1.

3.5 Legion progress

Legion is a set of LiRA components for data sharing and communication among Web clients. It allows programmers to design web applications where clients access a set of shared objects replicated directly at the client machines. The current Legion system assumes that applications have tens to a few hundred of users sharing the same state (i.e., the same data bucket).

The design and implementation of Legion is presented in Deliverable D5.1. The main contribution of Legion lies in its architecture and the combination of the multiple protocols that were used to achieve its design. The interested reader is therefore redirected to that document or to the original paper where this work was presented [9].

Shared objects are implemented as CRDTs, which guarantees consistency between clients. Clients can synchronize local replicas directly with each other, by leveraging on recent advances in browser technology, namely on Web Real Time Communication (WebRTC) connections. For ensuring durability of the application state, as well as to assist in other relevant aspects of the systems operation, Legion resorts to a set of centralized services. We designed Legion so that different Internet services (or a combination of Internet services and Legion's own support servers) can be employed. These services are accessed uniformly by Legion through a set of *adapters* with well defined interfaces.

We assume that the application state is organized in *containers*, that aggregate multiple data objects. Containers are replicated by clients completely. While Legion provides causal consistency guarantees over the state observed by clients over their local replicas, this is only provided for each container (i.e, causality is not enforced among different containers).

[1] https://github.com/SyncFree/antidote-fs

(a) Legion data types

Data objects in Legion are exposed to application programmers as CRDTs. The Legion runtime is responsible for managing the replication of these CRDTs among the clients (and the centralized infrastructure for a few clients as to ensure durability).

To support this programming model Legion provides an extensible library of data types. Objects are exposed to the application through (transparent) object handlers that hide the internal CRDT representation.

The CRDT library supports the following data types: Counters, Strings, Lists, Sets, and Maps. Our library uses Δ-based CRDTs [10], which are very flexible, allowing replicas to synchronize by using deltas with the effects of one or more operations, or the full state. This new type of delta-based CRDTs [1] is specially designed to allow efficient synchronization in epidemic settings, by avoiding, most of the times, a full state synchronization when two replicas connect for the first time. Each data type includes type-specific methods for querying and modifying its internal state, and generic methods to compute and integrate deltas (*i.e.,* differences among pairs of replicas).

In order to synchronize different replicas of an object, Legion relies on an application level multicast primitive that operates on top of the unstructured overlay that supports the replication of the corresponding data container.

The multicast primitive is used to propagate and receive deltas that encode modifications to the state of local replicas in a way that respects causal order (of operations encoded in these deltas). To achieve this, we use the following approach.

For each container, each client maintains a list of received deltas. The order of deltas in this list respects causal order. A client propagates, to every client it connects to, the deltas in this list respecting their order. The channels established between two clients are FIFO, i.e., deltas are received in the same order they have been sent.

When a client receives a delta from some other client, two cases can occur. First, the delta has been previously received, which can be detected by the fact that the delta timestamp is already reflected in the version vector of the container. In this case, the delta is discarded. Second, the delta is received for the first time. In this case, besides integrating the delta, the delta is added to the end of the lists of deltas to be propagated to other peers.

The actual implementation of Legion only keeps a suffix of the list of deltas received. Note that, at the start of every synchronization step, clients exchange their current vector clocks, which allow them, in the general case where their suffix list of deltas is large enough to include the logical time of their peer replicas, to generate deltas for propagation that contain only operations that are not yet reflected in that peer's state.

However, when two clients connect for the first time (or re-connect after a long period of disconnection), it might be impossible (or, at least, inefficient) to compute the adequate delta to send to its peer. In this case the two clients will synchronize their replicas by using the efficient initial synchronization mechanism supported by Δ-based CRDTs. In this case, if only a delta has been received, it is added to the list of deltas for propagation to other nodes. If it was necessary to synchronize using the full state, then the client needs to execute the same process to synchronize with other clients it is connected to.

4 Software

The basic programming model is released as three software artefacts, namely Lasp, AntidoteDB, and Legion.

4.1 Lasp system

Documentation https://lasp-lang.org

Code repository https://github.com/lasp-lang

4.2 AntidoteDB system

Documentation http://antidotedb.org

Code repository https://github.com/SyncFree/antidote

4.3 Legion system

Documentation https://legion.di.fct.unl.pt/

Code repository https://github.com/albertlinde/Legion

5 State of the art

We compare the LightKone work in programming models for edge computing to the state of the art, and we explain the innovations. For state of the art comparisons regarding algorithmic support we refer to the deliverables D3.1, D5.1, and D6.1.

5.1 Summary of SOTA

We split the SOTA discussion into two parts: technical SOTA and architecture SOTA. The most important SOTA comparison is with respect to edge architectures, in the architecture SOTA, but it is useful also to provide a brief technical SOTA, that is independent of edge architectures. In the technical SOTA, which is explained in Section (a), we explain the technical innovations of LiRA in the area of programming models. In the architecture SOTA, which is explained in the rest of Section 5, we present the major edge architectures that exist currently and we explain the programming model innovations of LiRA with respect to these architectures.

(a) Technical SOTA

Before presenting the edge architectures, we briefly summarize the technical innovations of LightKone. In this area, the state of the art consists of the following:

- Eventual consistent key/value stores, of which a representative system is Cassandra.
- Key/value stores with support for CRDTs, of which a representative system is Riak.
- Communication software providing reliable broadcast ability on dynamic networks, of which representative systems are existing publish/subscribe systems.

The key/value stores mentioned above execute in cloud environments or environments close to cloud (high-performance nodes). There exist edge extensions to these environments, but they are purely interfaces: the reliability and computation is provided by the key/value stores and not in the edge extensions.

The convergent data store provided by LiRA extends eventually consistent stores with a convergence property, which is explained in Appendix A. Convergence is an innovation with respect to eventually consistent data stores. The convergent data store provided by LiRA, in the Lasp component, uses a highly resilient communication component, Partisan, that is able to provide connectivity and broadcast even with extremely high node turnover, because it is based on hybrid gossip. The ability to provide these abilities is an innovation with respect to existing communication software, none of which is currently based on hybrid gossip.

(b) Architecture SOTA

At the present time there exist several large consortia that have defined edge architectures and that provide platforms to realize part of these architectures. We present four edge architectures, namely OpenFog RA, Microsoft Azure IoT, Amazon IoT Greengrass, and ECC Edge Computing. We summarizes the programming models of these architectures and give the LightKone programming model innovations with respect to these programming models. For more detailed explanations of these architectures and their relationship to LiRA, we refer to Deliverable D3.1.

In all the edge architectures presented here, the system consists of a basic foundational layer (typically providing network interface, virtualization, basic storage primitives, basic security primitives), with a service layer on top. The applications live on top of the service layer and interface with the services that they need. For nontrivial applications, this puts the burden of service coordination on the developer, who must coordinate all used services and their APIs. LightKone provides several innovations compared to these architectures, which significantly increase functionality for edge applications while at the same time lightening the developer's burden.

OpenFog Reference Architecture The OpenFog RA is based on a layered structure with Node-level services on the bottom (network, compute, storage, etc.), and application support services on top. OpenFog advocates two distributed computing patterns, namely client-server computing with transactional databases and publish/subscribe messaging. No resilience or consistency management is part of the reference architecture; it must be provided by services.

Microsoft Azure IoT Microsoft Azure IoT is based on a data streaming model, with bidirectional data streams from IoT devices to cloud services, using intermediate gateways. Computation at the edge is limited to aggregation and compression; decision making is done in the cloud. As programming model, Azure IoT recommends using actor frameworks, which consist of a large number of independent concurrent actors collaborating with each other. The actor model is very similar to Erlang's process model, and is implicitly supported by LightKone. No resilience or consistency management is part of the reference architecture; it must be provided by services.

Amazon IoT Greengrass Amazon IoT Greengrass is based on containerized local execution of AWS Lambda functions in a collection of local nodes together called a Greengrass Core, providing local computation and storage ability that does not need Internet connectivity. Communication between local Cores, other local services, and cloud services, is done using publish/subscribe messaging. It is possible to synchronize a Core with the cloud, but this is not required for Core operation. No other resilience or consistency management is part of the reference architecture; it must be provided by services.

ECC Edge Computing ECC Edge Computing is a high-level model that is based on using knowledge in specific application domains, called "model-driven" edge computing. It is a layered structure where the central concept is the ECN (Edge Computing Node) with its basic functionality, organized in a virtualization layer. Services run on top of this layer. There is support for basic data consistency (mentioned in the document, but not elaborated on) in the service layer. Causal relationships among edge data are mentioned as important, but no architectural support is given for them. Other than these mentions of consistency, there is no other resilience or consistency management in the reference architecture.

5.2 Innovation with respect to SOTA

We compare the programming model of LiRA (LightKone Reference Architecture, defined in Deliverable D3.1), with the state of the art in edge architectures as summarized in the previous section.

(a) Main innovation

Compared to the edge architectures, the core innovation of LightKone is a *convergent data store*, which combines four abilities:

1. Resilient data storage (replicated key/value store, based on CRDTs).
2. Resilient communication (based on hybrid gossip).
3. Distributed consistency (based on convergent consistency of CRDTs, causal consistency support, transaction support, and Just-Right consistency support).
4. Dynamic network support (high churn of nodes joining and leaving).

LightKone provides a single programming model that combines these four abilities. This programming model is declined into three variant APIs, targeted toward three fundamentally different edge scenarios: one in heavy edge (a.k.a. fog computing)[2], and two in light edge (a.k.a. edge computing in the OpenFog RA document, as opposed to fog computing). These three variants are the following:

- The Antidote system, targeted toward heavy edge scenarios.
- The Lasp system, targeted toward light edge scenarios for sensor networks.
- The Legion system, targeted toward light edge scenarios for mobile networks (networks of communicating smartphone clients).

In addition to this fourfold core innovation, LightKone provides additional innovations that support this core for specific use cases. These additional innovations have little effect on the programming model, so they are described in the other deliverables, namely D3.1, D5.1, and D6.1.

(b) Elaboration of the innovations

In this section we give more information on the LightKone innovations. For full information, please see the software deliverables and the published documentation (including scientific papers).

- JRC methodology for heavy edge applications (see also Section 3.4). State of the art systems for heavy edge are designed primarily to support strong consistency (e.g., Spanner, and Cassandra in the appropriate configuration), and their support for weaker consistency is added for performance and does not guarantee application invariants. JRC innovates with respect to this by guaranteeing application invariants also in the case of weak consistency.

[2]The difference in terminology between the LightKone Grant Agreement and the OpenFog RA occurs because the OpenFog RA document was published after the start of LightKone.

- Semantics for the reference architecture (see also Section 3.3 and Appendix A). Three important components of LiRA, namely Antidote, Legion, and Lasp, are based on extensions of the CRDT concept. The existence of a semantics for Antidote and Lasp is an innovation with respect to state of the art. State of the art systems do not have a formal semantics. The semantics is important for us, however, because it guarantees consistency for full programs and not just single CRDTs. In particular, the semantics guarantees a smoother improvement path for LiRA and a reduced likelihood for unpleasant surprises as new functionality is added. This is important since we are breaking new ground; the semantics helps us reduce risks when building innovative systems.

- Operational semantics for Antidote (see also Section (b) and Appendix B). No state of the art database has a documented formal operational semantics; rather the semantics is given informally in natural language. The existence of this semantics is an innovation that is necessary to prepare future evolution of the Antidote component in LiRA. The semantics is a necessary prerequisite for tool support and for supporting powerful methodologies such as JRC.

- Materialized views for light edge applications (see Section (b)). The ability to link CRDTs together in Lasp provides materialized views, which is an approach to perform computation while guaranteeing consistency. Because of the semantics, it guarantees convergence, reliability, and scalability for such computations, just as for the CRDTs themselves. All existing CRDT-based databases in the state of the art (of which the most visible example is Riak) are limited to CRDT-based data storage without support for transactions or materialized views or causality. LightKone innovates by adding these abilities to CRDT data storage.

- Peer-to-peer CRDTs for mobile applications. The use of CRDTs directly in mobile clients is an innovation with respect to state of the art. No existing state of the art system, to our knowledge, runs CRDTs directly on mobile devices. This innovation guarantees data consistency for data stored in the mobile phone.

- Resilient communication for dynamic networks. The Partisan library provides a reliable broadcast operation for highly dynamic networks, by using hybrid gossip to reorganize the communication structure on the fly. The use of hybrid gossip on edge networks is an innovation with respect to state of the art ad hoc networks, that provides much increased resilience. Partisan's internal hybrid gossip algorithms, namely Plumtree and HyParView, main connectivity with high probability despite losing 90% of nodes.

6 Exploratory work

Exploratory work is an important part of a European-funded RIA because it investigates and develops ideas that advance the state of the art both in research and industry. This section explains the exploratory work in the area of programming models that has been done during the first year and gives indications how to decide whether the work will make it into a future version of LiRA. It is clear that LiRA will always be evolving, since edge computing is an evolving field. The exploratory work prepares for LiRA's future development. This work is risky in the sense that not all of it will become part of the reference architecture. Such risk-taking is a necessary part of all successful research. Success of this work is to be measured not on how much of the work becomes part of the reference architecture, but on whether sufficiently innovative exploration is done and on whether the reference architecture itself is sufficiently innovative.

Since LightKone is an RIA with both a research and an innovation part, the research part must be active to explore new ideas that can migrate into the innovation part. A second reason for the exploratory work is as preparation for continued research after the LightKone project ends. Research is a continuous activity that should not be interrupted by project boundaries, if we want to maximize its productivity. Future innovation done by the partners, possibly beyond LightKone, depends also on exploratory activity done within LightKone. A third reason for the exploratory work is to improve our understanding of the area, to indicate the way for relevant future innovations.

6.1 Available file system

The basis for state of the art file systems is the Posix semantics, which is designed under the assumptions of strong consistency and synchrony. This is safe but it underperforms at large scale and it is unavailable during network partitioning. Real-world experience shows that the synchrony assumption is too strong: concurrent updates to the same file system objects are rare. Therefore, state of the art file systems, such as HDFS, NFS, and PVFS, eschew the synchrony assumption and replace it by asynchronous replication. However, these file systems may in some cases violate integrity invariants.

The file system we present here innovates with respect to the state of the art by guaranteeing integrity invariants while at the same time supporting asynchronous replication. Our design is as asynchronous as possible while satisfying the invariants. We provide two solutions beyond the state of the art:

- Fully asynchronous file system, which accepts all concurrent updates but weakens sequential Posix semantics by duplicating directories that would otherwise end up in a cycle.

- Mostly-asynchronous file system, in which most operations run asynchronously, and only the move-directory operations might be blocked by synchronization.

This work is presented in the VMCAI 2018 paper (see Section 7.1). The work has several goals. First, it is a practical application of the JRC methodology, and is used to strengthen this methodology. Second, it is an extension of the state of the art in Posix-based file system semantics, and as such can potentially be a useful LiRA component.

6.2 Quality-aware reactive programming

Quality-Aware Reactive Programming (Quarp) is a dataflow language for distributed components, which is a variant of a *reactive programming* model tailored for lightweight distributed systems (see FSEN 2017 paper in Section 7.1). In the reactive paradigm concurrent objects can produce values that trigger the execution of other objects, in a dynamic dependency graph. Shifting this paradigm from locally concurrent objects to concurrent components brings new challenges when managing consistency (called glitch-freedom in this context) without incurring large performance overheads.

The key problem addressed by Quarp is how to optimise the reactive paradigm to lightweight distributed components, i.e., systems where communication is unreliable and where each node has a limited amount of memory space, computational power, and energy. Our approach enriches messages with contextual information about the whereabouts of the data producers – with properties that allow these to be combined and compared, and used as a local quality metric. This contextual information indicates components whether the received inputs have *enough quality* (or enough compatibility) to be used in their computation, and consequently to produce a new value to their subscribers.

In the context of LightKone, this paradigm fits well the light-edge scenarios. Moreover, the existing theory and implemented libraries for managing (eventual) data consistency can form the basis of an implementation for quality-aware reactive systems. For example, to optimise multicast communications, and to better maintain global quality of a running system by iteratively adapting the local quality parameters of each component. More information on the principles behind Quarp and on the integration of data consistency follow below.

6.3 Rethinking distributed programming

The LightKone innovations are steps in the direction of abstracting away part of the difficulty of building distributed systems, leaving visible only the essential aspects that must remain visible. We are exploring going even further in this direction, to understand how to strengthen the LightKone innovations. Here we present three explorations of future distributed systems that can influence future LightKone developments.

(a) Lasp as a service composition language

Lasp can be seen as a service composition language. This is a way to use legacy libraries for distributed services (e.g., consensus, analytics, communication, etc.) as part of the run-time for Lasp. This solves the greenfield problem for edge computing: it is not possible for us to reimplement all the legacy support for distributed computation, but ideally we should be able to reuse it. The existing Lasp system is a first step toward this solution: it composes several libraries including communication and orchestration.

This idea is expressed in two documents: a paper presented at the OBT 2018 workshop (see Section 7.2) and a keynote talk at the Velocity 2017 conference (see Section 8.1). The keynote defines a hypothetical language called Martinelli that realizes the service composition idea. Instead of having to compose legacy libraries by hand, which is what developers currently do, the Martinelli run-time system does this composition automatically. This simplifies developers' work while guaranteeing better semantics for the whole.

(b) Edge computation as decentralized truth

In the traditional view of distributed applications, the contents of a centralized database is considered to be the truth, i.e., the definitive correct data for the application. In reality, however, the truth is not in the database, but distributed among the clients (at the edge), and the database reflects this information with some staleness. We propose therefore to invert the traditional view and to consider the database as an optimization for the true data at the edge. This idea is expressed in the Salon des Refusés 2017 paper (see Section 7.2).

Even though many applications treat the database contents as the truth, the database is in fact just an optimization that approximates the correct data. We can say that the database is eventually consistent with respect to client data. The LightKone vision of convergent data supports this inversion. The LiRA programming model and its implementations support eventual consistent computation with convergence properties (i.e., divergence with the correct result always tends toward zero). In particular, our work on heavy edge computing, based on the Antidote database, can directly support this new approach.

(c) Distributed programming in the context of CAP

It is useful to explore the space of possible distributed programming models in the context of the CAP theorem. The CAP theorem places rigorous limitations on the power of distributed programming models. It shows that there are two extreme points in the spectrum of these models: AP (available and partition-tolerant, but not consistent), and CP (consistent and partition-tolerant, but not available). Distributed programming models can be classified on a spectrum from AP on one side to CP on the other side. This idea is expressed in the PMLDC 2017 paper (see Section 7.2).

The Lasp programming model is AP: it is always possible to do computations locally even when there are partitions, but there can be divergence between nodes. The Lasp semantics guarantees that node data will converge to a consistent state. This suggests that we can imagine another model, called Austere in the paper, at the other side of the spectrum, namely it is CP. In a synthesis of both extremes, we combine both Lasp and Austere to define a model called Spry, in which the trade-off between availability and consistency can vary at different parts of the application.

7 Published papers

7.1 Refereed conference papers

- José Proença and Carlos Baquero. *Quality-Aware Reactive Programming for the Internet of Things*, 7th IPM International Conference on Fundamentals of Software Engineering (FSEN 2017), Tehran, Iran, April 26-28, 2017.

- Christopher Meiklejohn, Vitor Enes, Junghun Yoo, Carlos Baquero, Peter Van Roy, and Annette Bieniusa. *Practical Evaluation of the Lasp Programming Model at Large Scale*, 19th International Symposium on Principles and Practice of Declarative Programming (PPDP 2017), Namur, Belgium, Oct. 9-12, 2017.

- Mahsa Najafzadeh, Marc Shapiro, and Patrick Eugster. *Co-Design and Verification of an Available File System.* 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2018), Los Angeles, CA, Jan. 7-13, 2018.

7.2 Refereed workshop papers

- Christopher Meiklejohn. *A Certain Tendency of the Database Community*, Salon des Refusés workshop (colocated with <Programming> 2017), Brussels, Belgium, April 3-6, 2017.
- Christopher Meiklejohn. *On the Design of Distributed Programming Models*, Second Workshop on Programming Models and Languages for Distributed Computing (PMLDC 2017) (colocated with ECOOP 2017), Barcelona, Spain, June 20, 2017.
- Christopher Meiklejohn and Peter Van Roy. *Towards a Systems Approach to Distributed Programming*, Off the Beaten Track workshop (OBT 2018, colocated with POPL 2018), Los Angeles, CA, Jan. 13, 2018.

8 Other dissemination

8.1 Invited talks

- Peter Van Roy. *Ditching the Data Center: How to Stop Worrying and Love the Edge*, Erlang User Conference, Stockholm, Sweden, June 8-9, 2017.
- Christopher Meiklejohn. *Scaling a Startup with a 21st Century Programming Language*, Velocity 2017 Systems Engineering Conference, London, UK, Oct. 18-20, 2017.

8.2 Submitted papers

- Marc Shapiro, Annette Bieniusa, Nuno Preguiça, and Christopher Meiklejohn. *Just-Right Consistency*, Submitted to CACM, 2017.
- Kevin Clancy, Heather Miller, and Christopher Meiklejohn. *Monotonicity Types*, Submitted to ESOP 2018.

A Unified semantics for LiRA (Antidote and Lasp)

Unified Semantics for Lightkone (Antidote $\cup$ Lasp)

Peter Zeller
Annette Bieniusa
Mathias Weber
Christopher Meiklejohn
Peter Van Roy
Nuno Preguiça
Carla Ferreira

January 10, 2018

1 Introduction

This document defines the first version of a formal abstract semantics for a general computation model for synchronization-free computing. The semantics is based on the event visibility formalism of Sebastian Burckhardt [1] and subsumes the Lasp and Antidote computation models defined in the SyncFree project. This is a significant step toward achieving a major goal of the LightKone project, namely defining a single computation model that encompasses the full spectrum of synchronization-free computation.

In SyncFree we explored how to build large distributed systems where the basic data structure is the CRDT, Conflict-Free Replicated Data Type, which is a distributed data structure with the property that replicas eventually become consistent with each other. CRDTs are based on a very weak synchronization model, namely eventual replica-to-replica communication. No other synchronization is needed. We defined two very different computation models and their implementations, namely Lasp and Antidote, that are both based on CRDTs. Lasp is a dataflow computation model based on CRDT composition, i.e., it uses dataflow communication between CRDT instances. Antidote is a causal transactional database where the basic data structure is the CRDT. Lasp runs naturally on an edge network. Antidote runs naturally on a georeplicated data center.

2 Abstract executions

We describe the system based on the events which happened on the system. Each event has an operation $op \in Ops$ and a result value $r \in \mathcal{V}$. Each event works on one specific object identified by a key $k \in Keys$. The key contains the type of the object ($type(k)$). Moreover, an event can carry meta-data like a version or a transaction-identifier, which we use to describe optional guarantees of our unified system model.

2.1 Internal view

To specify which executions are allowed, we use relations vis and ar. These relations are not directly observable by clients, but we consider them part of the execution.

An **execution** is a tuple $(E, key, op, res, vis, ar)$ where

- E is a finite set of events.
- key assigns a key to each event in E.
- op assigns an operation to each event in E.
- res assigns a result value to each event in E.
- vis is the visibility relation on events in E. We write $e_1 \prec_{vis} e_2$ for $(e_1, e_2) \in vis$. Intuitively, $e_1 \prec_{vis} e_2$ means, that e_1 happened before e_2 and e_2 can thus observe the effects of e_1.
- ar is the arbitration relation on events in E. We use it to break ties between concurrent updates. In the implementation we mostly use extended timestamps.

The behavior of data types T is specified by a function $\mathcal{F}_T$ which takes an operation context and returns a result value. An operation context is a tuple (E, op, vis, ar) where

- E is the set of events on same key and in visibility
- $args$ give the operation arguments for each event in E
- vis and ar are the relations from the execution restricted to the events in E

To get the operation context for an event e we use the function $ctxt(e)$, which is defined as: $ctxt(e) = \left(E', op_{|E'}, vis_{|E'}, ar_{|E'}\right)$ where $E' = \{e' \in E \mid e' \prec_{vis} e\}$.

If c is a context, then $c_{|k}$ is the context restricted to key k, which is defined as: $(E, op, vis, ar)_{|k} = \left(E', op_{|E'}, vis_{|E'}, ar_{|E'}\right)$ where $E' = \{e \in E \mid key(e) = k\}$.

2.2 Conditions

An execution is correct, if it satisfies the following conditions:

Acyclic visibility vis is acyclic

Arbitration is total ar is a total order

Eventual consistency There can only be a finite set of events on the same object that not see a certain event.

$\forall e \in E.\ \mathit{finite}(\{e' \in E \mid \mathit{key}(e) = \mathit{key}(e') \wedge e \not\prec_{vis} e'\})$

In addition to the conditions above, some Lightkone system satisfy some of the following guarantees:

2.2.1 Key value store:

In a key-value store, the result of each read event can be explained by the operation context of the read-operation:

Correct results (KV-Store) For all $e \in E$ we have $\mathit{res}(e) = F_{\mathit{type}(\mathit{key}(e))}\left(\mathit{op}(e), \mathit{ctxt}(e)_{|\mathit{key}(e)}\right)$

2.2.2 Causality:

Per object causal consistency for all keys k: $\mathit{vis}_{|\{e \mid \mathit{key}(e)=k\}}$ is transitive.

Causal consistency vis is transitive.

2.2.3 Transactions:

To model transactions, we assume that we know the transaction each event originated from: $\mathit{tx}(e)$ is the originating transaction of event e. We assume that all transactions are committed. This is possible because we do not support isolation levels, where uncommitted transactions can have any affect on events outside the transaction, and we do not include time in our model.

Atomic visibility For events $e_1, e_1', e_2, e_2' \in E$ with $\mathit{tx}(e_1) = \mathit{tx}(e_1') = \mathit{tx}_1$ and $\mathit{tx}(e_2) = \mathit{tx}(e_2') = \mathit{tx}_2 \neq \mathit{tx}_1$ we have $e_1 \prec_{vis} e_2 \leftrightarrow e_1' \prec_{vis} e_2'$.

Sequential transactions For all transactions t the relation $\mathit{vis}_{|\{e|\mathit{tx}(e)=t\}}$ is a total order.

2.2.4 Versioned Store:

For this extension, we assume that users can provide a version for each event ($\mathit{version}(e)$). If no version is given by a user, $\mathit{version}(e) = \bot$. Otherwise we will treat a version as a set of events.

Min snapshot For all events e, e': $e' \in \mathit{version}(e)$ implies $e' \prec_{vis} e$.

Precise snapshot For all events e, e': $e' \in \mathit{version}(e)$ iff $e' \prec_{vis} e$.

2.2.5 Session guarantees

We assume that events issued from the same session (in practice this can be a database connection) are ordered by a session order *so* on events. $e_1 \prec_{so} e_2$ if e_1 was submitted before e_2 on the same session.

To define the classical session guarantees, we need to distinguish read and write operations. We use the predicates $\mathit{isRead}(e)$ and $\mathit{isWrite}(e)$ to do so.

Read Your Writes $e_1 \prec_{so} e_2 \wedge \mathit{isWrite}(e_1) \wedge \mathit{isRead}(e_2) \longrightarrow e_1 \prec_{vis} e_2$

Monotonic Reads $e_1 \prec_{so} e_2 \wedge \mathit{isRead}(e_1) \wedge \mathit{isRead}(e_2) \longrightarrow (\forall e'.\ e' \prec_{vis} e_1 \rightarrow e' \prec_{vis} e_2)$

Writes Follow Reads $e_1 \prec_{so} e_2 \wedge \mathit{isRead}(e_1) \wedge \mathit{isWrite}(e_2) \longrightarrow (\forall e'.\ e' \prec_{vis} e_1 \rightarrow e' \prec_{vis} e_2)$

General session guarantee $e_1 \prec_{so} e_2 \longrightarrow e_1 \prec_{vis} e_2$

2.2.6 Lasp store:

To specify the Lasp semantics, we use additional meta data:

Lasp Objects We partition the key space into Lasp-objects and base-objects. The set $\mathit{LaspKeys} \subset \mathit{Keys}$ specifies, which keys identify Lasp-objects. All other keys refer to base-objects.

Clients can only perform updates on base-objects, but they can read from both kind of objects.

Link The function *link* specifies a Lasp-link for each Lasp-key. A Lasp-link consists of a list of n keys, and a function f, which takes n values and returns a single value.

To give a better intuition about links, we give some examples of common operations.

map Let k be a key identifying an object with a set-value containing elements of type T and f be a function $T \rightarrow S$ for some type S. Then $\mathit{map}(k, f)$ is the link reading the set from object k and applying f to all elements in the set.

$\mathit{map}(k, f) := ([k], \lambda V \rightarrow \{f(x) \mid x \in V\})$

product Let k_1 and k_2 be keys identifying objects with set-values. Then $\mathit{product}(k_1, k_2)$, which reads from both keys and yields the Cartesian product of the two sets.

$\mathit{product}(k_1, k_2) := ([k_1, k_2], (\lambda V_1, V_2 \rightarrow V_1 \times V_2\}))$

intersection $\mathit{intersection}(k_1, k_2) := ([k_1, k_2], (\lambda V_1, V_2 \rightarrow V_1 \cap V_2)\})$

union $\mathit{union}(k_1, k_2) := ([k_1, k_2], (\lambda V_1, V_2 \rightarrow V_1 \cup V_2)\})$

filter $\mathit{filter}(k, P) := ([k], \lambda V \rightarrow \{x \mid x \in V \wedge P(x)\})$

fold Let k be a key identifying an object with a value of type $T\ \mathit{set}$, $f : T \times S \rightarrow S$ be a function, and z an initial value of type S.

$\mathit{fold}(k, f, z) := ([k], \mathit{fold}_{f,z})$

where $fold_{f,z}\{\} = z$ and $fold_{f,z}(\{x\} \cup V) = f(x, fold_{f,z}(V))$.

The function f and value z have to be chosen, such that the order of evaluation does not matter.

Convergence of Linked objects If a Lasp-object k_1 depends on a base-object k_2, then the system ensures eventual consistency between the two. This means that an update k_2 cannot be invisible to an infinite number of read events on k_1.
Formally, the set $\mathit{dependsOn}(k)$ for Lasp-objects k is inductively defined:

- Direct dependencies: If $\mathit{link}(k) = (K, f)$ then $\mathit{set}(K) \subseteq \mathit{dependsOn}(k)$.
- Transitivity: If $k_1 \in \mathit{dependsOn}(k2)$ and $k_2 \in \mathit{dependsOn}(k_3)$, then $k_1 \in \mathit{dependsOn}(k_3)$.

Then eventual consistency of Links is defined by the condition:
$\forall e \in E.\ \mathit{finite}(\{e' \in E \mid \mathit{key}(e) \in \mathit{dependsOn(key(e'))} \wedge e \not\rightarrow_{vis} e'\})$

Reading from Lasp objects We now define how the result of read-operations is determined. To do this, we define a function R, which takes a key k and an operation context and returns the result value for the read operation.
For all $e \in E$ we have $\mathit{res}(e) = R(\mathit{key}(e), \mathit{ctxt}(e))$.
The function R is defined recursively and distinguishes two cases:

- For reads on base-objects the semantics are determined by the datatype specification, similar to the correctness condition of the key-value store (see 2.2.1):
  If $k \notin \mathit{LaspKeys}$, then $R(k, \mathit{ctxt}) = F_{\mathit{type}(k)}\left(\mathit{ctxt}_{|k}\right)$
- For reads on Lasp-objects, the result is determined by the link assigned to the respective object.
  If $\mathit{link}(k) = ([k_1, \ldots, k_n], f)$, then $R(k, \mathit{ctxt}) = f(R(k_1, \mathit{ctxt}), \ldots, R(k_n, \mathit{ctxt}))$.

Cycles This definition can be used, if there are no cycles in the link-graph (or infinite sequences in the link-graph). To support cycles, we determine the result by taking the least fixed point of R's equations.
To ensure that the least fixed point is well-defined, we assume that all values are part of a complete lattice and we slightly adapt the equations for R to make them monotonic:

- If $k \notin \mathit{LaspKeys}$, then $R(k, \mathit{ctxt}) = F_{\mathit{type}(k)}\left(\mathit{ctxt}_{|k}\right)$
- If $\mathit{link}(k) = ([k_1, \ldots, k_n], f)$, then $R(k, \mathit{ctxt}) \supseteq f(R(k_1, \mathit{ctxt}), \ldots, R(k_n, \mathit{ctxt}))$.

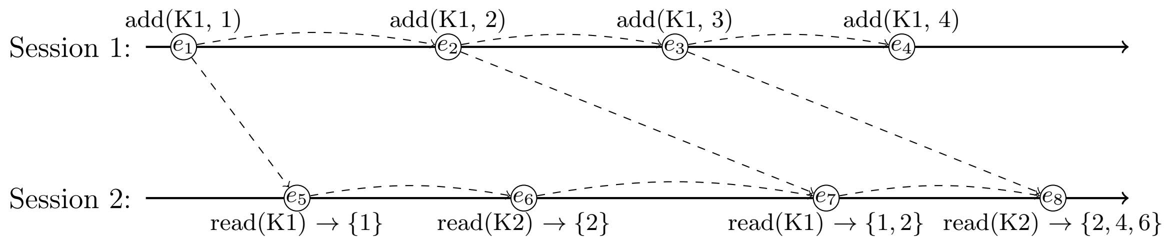

Figure 1: Example 1: Abstract execution of a Lasp program

2.3 Example: A Lasp program

Consider the following Lasp program:

```
{ok, {K1, _, _, _}} = lasp:declare({<<"k1">>, state_orset}, state_orset),
{ok, {K2, _, _, _}} = lasp:declare({<<"k2">>, state_orset}, state_orset),
{ok, _ } = lasp:map(K1, fun(X) -> X * 2 end, K2).
```

This defines a base object with key k_1 and a Lasp object with key k_2 and $link(k_2) = map(k_1, \lambda x \to x \cdot 2) = ([k_1], \lambda V \to \{x \cdot 2 \mid x \in V\})$.

Execution 1 The graph in Figure 1 shows an example of an abstract execution for this Lasp program. Sessions are drawn as horizontal solid arrows and visibility between events is visualized with dashed arrows. This example assumes causal consistency, so we omit transitive visibility edges in the graph.

We now use the formal definition to explain why $res(e8) = \{2, 4, 6\}$.

- First we calculate the execution context of e_8.

  The set of visible events is $E' = \{e_1, e_2, e_3, e_5, e_6, e_7\}$.

  Then $op_{|E'} = \{e_1 \mapsto add(1), e_2 \mapsto add(2), e_3 \mapsto add(3), e_4 \mapsto read(), e_5 \mapsto read(), e_6 \mapsto read(), e_7 \mapsto read()\}$ and $vis_{|E'} = \{(e_1, e_2), (e_1, e_5), (e_2, e_3), (e_2, e_7), (e_5, e_6), (e_6, e_7)\}^+$. The arbitration relation ar is not relevant for this example.

  $ctxt(e_8) = (E', op_{|E'}, vis_{|E'}, ar_{|E'})$

- The result is specified by the R function: $res(e_8) = R(k_2, ctxt(e_8))$.

- Because $k_2 \in LaspKeys$ and $link(k_2) = ([k_1], \lambda S \to \{x \cdot 2 \mid x \in S\})$, we have:

  $R(k_2, ctxt(e_8)) = (\lambda S \to \{x{\cdot}2 \mid x \in S\})(R(k_1, ctxt(e_8))) = \{x{\cdot}2 \mid x \in R(k_1, ctxt(e_8))\}$

- As k_1 is an add-wins set, we have:

  $R(k_1, ctxt(e_8)) = \mathcal{F}_{aw\text{-}set}(ctxt(e_8))$
  $= \{x \mid \exists a \in E'.\ op(a) = add(x) \land \nexists r \in E'.\ op(r) = remove(x) \land a \prec_{vis} r\}$
  $= \{1, 2, 3\}$

- With this we get, that $R(k_2, ctxt(e_8)) = \{x \cdot 2 \mid x \in \{1, 2, 3\}\} = \{2, 4, 6\}$.

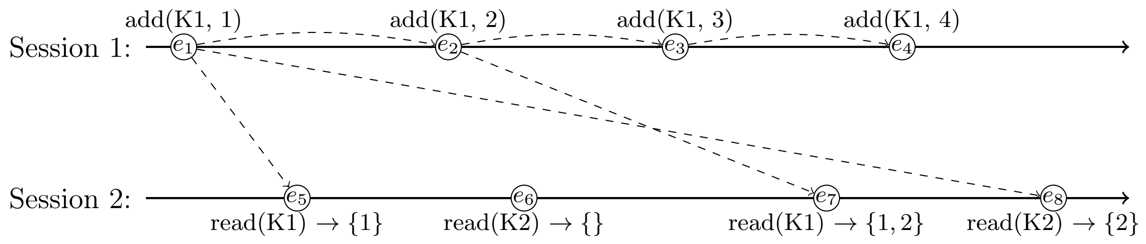

Figure 2: Example 2: Abstract execution of a Lasp program

Execution 2 In contrast to the first execution, the graph in Figure 2 does not assume session guarantees and thus allows the reads on K2 to observe fewer events than previous reads did. Because session guarantees are not included, it is not necessary that e_4 is visible to e_5, even though they happened on the same session. Therefore $e5$ can observe a state, where e_1 is not yet visible.

2.4 Example: Antidote

Executions on Antidote satisfy all 3 basic definitions and provide the following additional guarantees:

- Causal consistency
- Atomic visibility
- Min snapshot (When starting a transaction, a minimum snapshot version can be provided. On commit clients receive a version including the committed transaction).

Antidote data types: To specify the antidote data types, we define the following auxiliary functions:

$$\begin{aligned}
\mathit{filterResets}(E, op, vis) &= \{e \in E \mid op(e) \neq \mathit{reset}() \land \nexists r \in E.\ op(r) = \mathit{reset}() \land e \prec_{vis} r\} \\
\mathit{latestEvents}(E, vis) &= \{e \in E \mid\ \nexists e' \in E.\ e \prec_{vis} e'\} \\
\mathit{max}(E, ar) &= \text{the } e \in E\text{, such that } \forall e' \in E. e' \prec_{ar} e
\end{aligned}$$

With these functions we can define the semantics of the different data types:

$$\begin{aligned}
\mathcal{F}_{\mathit{lww\text{-}reg}}(E, op, vis, ar) = \ & \textbf{let } E' = \mathit{latestEvents}(\mathit{filterResets}(E, op, vis), ar) \textbf{ in} \\
& \textbf{if } E' = \emptyset \textbf{ then } \texttt{""} \\
& \textbf{else } = \text{the } v\text{, such that } op(\mathit{max}(E', ar)) = \mathit{assign}(v)
\end{aligned}$$

$$\mathcal{F}_{\mathit{counter}}(E, op, vis, ar) = \sum_{(e,v).\ \exists e \in E.\ op(e)=increment(v)} v - \sum_{(e,v).\ \exists e \in E.\ op(e)=decrement(v)} v$$

$$\mathcal{F}_{\mathit{fat\text{-}counter}}(E, op, vis, ar) = \mathcal{F}_{\mathit{counter}}(\mathit{filterResets}(E, op, vis), op, vis, ar)$$

$$\begin{aligned}\mathcal{F}_{\mathit{ew\text{-}flag}}(E, op, vis, ar) = \ & \textbf{let } E' = \mathit{latestEvents}(\mathit{filterResets}(E, op, vis), ar) \textbf{ in}\\ & \exists e \in E'.\ op(e) = enable()\end{aligned}$$

$$\begin{aligned}\mathcal{F}_{\mathit{dw\text{-}flag}}(E, op, vis, ar) = \ & \textbf{let } E' = \mathit{latestEvents}(\mathit{filterResets}(E, op, vis), ar) \textbf{ in}\\ & \exists e \in E'.\ op(e) = enable() \wedge \nexists d \in E'.\ op(e) = disable()\end{aligned}$$

$$\mathcal{F}_{\mathit{g\text{-}set}}(E, op, vis, ar) = \{x \mid \exists a \in E.op(a) = add(x)\}$$

$$\begin{aligned}\mathcal{F}_{\mathit{aw\text{-}set}}(E, op, vis, ar) = \ & \textbf{let } E' = \mathit{filterResets}(E, op, vis) \textbf{ in}\\ & \{x \mid \exists a \in E'.\ op(a) = add(x) \wedge \nexists r \in E'.\ op(r) = remove(x) \wedge a \prec_{vis} r\}\end{aligned}$$

$$\begin{aligned}\mathcal{F}_{\mathit{rw\text{-}set}}(E, op, vis, ar) = \ & \textbf{let } E' = \mathit{filterResets}(E, op, vis) \textbf{ in}\\ & \{x \mid (\exists a \in E'.\ op(a) = add(x))\\ & \quad \wedge \forall r \in E'.\ op(r) = remove(x) \rightarrow \exists a \in E'.\ op(a) = add(x) \wedge r \prec_{vis} a\}\end{aligned}$$

3 Concrete Executions

To reason about correctness of algorithms and protocols with respect to the abstract semantics defined above, we use concrete executions. Here, we consider the concrete state of nodes in the system and messages passed between nodes.

3.1 Background

In this section we summarize the general framework for modeling concrete executions. The model is very close to Burckhardt's model.

An implementation consists of several role instances, which represent a kind of actor which can process client requests sequentially. Some role instances will be internal and do not process any requests from clients. Role instances can communicate by sending messages and they may contain active processes to execute actions periodically.

3.1.1 Role automata

The behavior of a single role instance is defined by a role automaton. A **role automaton** is a set of transitions over a set of operations O, a set of values $\mathcal{V}$ a set of states Σ, a set of messages M and a set of processes P.

Possible transitions are:

- $\mathit{init}(\sigma', M_{\mathit{sent}})$ with $\sigma \in \Sigma$ and $M_{\mathit{sent}} \subseteq M$
  Initializes the role instance.
- $\mathit{call}(o, \sigma, \sigma', M_{\mathit{sent}}, v)$ with $o \in O$, $\sigma, \sigma' \in \Sigma$, $M_{\mathit{sent}} \subseteq M$, and $v \in (\mathcal{V} \cup \bot)$
  Reacts to a call of operation o from a client.
- $\mathit{rcv}(m, \sigma, \sigma', M_{\mathit{sent}}, v)$ with $m \in M$, $\sigma, \sigma' \in \Sigma$, $M_{\mathit{sent}} \subseteq M$, and $v \in (\mathcal{V} \cup \bot)$
  Reacts to receiving message m.
- $\mathit{step}(p, \sigma, \sigma', M_{\mathit{sent}}, v)$ with $p \in P$, $\sigma, \sigma' \in \Sigma$, $M_{\mathit{sent}} \subseteq M$, and $v \in (\mathcal{V} \cup \bot)$
  Actively do a step in process p.

In the following we use functions to select certain information from a transition t: $\mathit{op}(t) = o$, $\mathit{rcv}(t) = m$, $\mathit{proc}(t) = p$, $\mathit{pre}(t) = \sigma$, $\mathit{post}(t) = \sigma'$, $\mathit{snd}(t) = M_{\mathit{sent}}$, and $\mathit{rval}(t) = v$. If the respective information does not exist for transition t the function returns $\bot$.

The set M_{sent} is a set of messages sent to all other role instances. The value v is the value returned to the last client call or $\bot$ if there is no return value. A role automaton can do several internal steps before replying to a client, for example it can send messages to other instances and wait for responses before replying to the client.

A role automaton must satisfy the following properties:

(r2) There is at least one initialization transition.

(r3) All messages can be received in all states. (Role automata can ignore unwanted messages or store them in their state for later processing.)

(r4) All operations can be called in all states.

(r5) All processes can take steps in all states.

3.1.2 Trajectories

A trajectory is a tuple $(E, \mathit{eo}, \mathit{tr})$ and models an execution of a single role automaton. E is a set of events, eo is a total order on these events and tr is the transition of each event.

As the events are totally ordered by eo, we can define what the predecessor of an event is. The event without predecessor must have an init transitions. For all other events e_i with predecessor e_{i-1}, the pre- and post-states must match: $\mathit{pre}(\mathit{tr}(e_i)) = \mathit{post}(\mathit{tr}(e_{i-1}))$

Moreover, there can be at most one pending call at a time, and events can only have a result value if there is a pending call.

3.1.3 Concrete Executions

Having defined possible trajectories for describing possible behaviors of a single role automaton instance, we can now define concrete executions, which combine a (possibly infinite) set *Roles* of such instances.

A concrete execution is a tuple $(E, eo, tr, role, del)$, such that:

(c1) eo is a total order on E.

(c2) tr specifies the transition of each event.

(c3) $role$ specifies a role from *Roles* for each event.

(c4) For each role r, the tuple (E_r, eo, tr) is a valid trajectory with respect to the automaton of r. Here E_r is the events in instance r: $E_r = \{e \in E \mid role(e) = r\}$.

(c5) del is a relation on events describing message delivery.

If $s \xrightarrow{del} r$, then $s \xrightarrow{eo} r$, $rcv(r) \neq \bot$, and $rcv(r) \in snd(s)$.

We also write $s \xrightarrow{del(m)} r$ to denote that message m has been sent from s to r, i.e. $s \xrightarrow{del} r \wedge rcv(r) = m$.

3.1.4 Transport guarantees

Similar to the conditions for abstract executions (see Section 2.2), we can define message delivery guarantees, which we can assign to certain classes of messages and which must be guaranteed by the underlying network stack.

- dontforge
- dontduplicate
- dontlose
- pairwiseordered
- reliable, reliablestream
- eventual
- eventualindirect

3.1.5 From concrete to abstract executions

Given a concrete execution, we can derive an observable history, by only considering events related to calls from client.

If there exist vis and ar relations, such that the observable history can be extended to a valid abstract execution, then the the concrete execution is correct.

3.1.6 Describing protocols

3.2 Protocols

Following Burckhardt's verification approach, we define protocols for the different actors (or: roles) in the system. Roles are defined as state machines. Interaction between roles involves the sending and receiving of messages, including the required message delivery guarantees.

We show here as first example the protocol for a causally consistent CRDT store (adapted from Burckhardt Fig. 6.13, there named CausalStreams).

```
protocol CausalCRDTStore< ⟨𝓕⟩ {
  // vector clock
  type VClock = map⟨int,int⟩;
  // returns maximal entry of vc
  function max(vc: VClock) { return vc.values().max(); }

  // updates
  struct Update(op: Operation, vc: VClock)
  // max entry in vector clock reveals sender of update
  function origin(u: Update) { return u.vc.key_of_max_value; }
  function visibleTo(vc1: VClock, vc2: VClock) {
    return ∀ i: vc1[i] ≤ vc2[i];
  }
  function arbitedBefore(1: Update, u2: Update) {
    return max(u1.vc) < max(u1.vc)
        || max(u1.vc) = max(u2.vc) && origin(u1.vc) < origin(u2.vc); }
  }

  message Notify(update: Update, vc: VClock) : reliablestream

  role Replica(pid: int) {
    var known: set⟨Update⟩;
    var vc: VClock;
    var pending: map⟨nat, queue⟨Update⟩⟩;

    // client issues operation
    operation perform (op: Operation) {
      // calculate return value based on local context
      var rval = 𝓕(op, makecontext(known, arbitedBefore, visibleTo));
      // advance logical clock
      vc[pid] = max(vc) + 1;
      // contruct update struct
      var u = Update(op, vc);
      // add to locally known updates
      known.add(u);
      // forward to other replicas
      send Notify(u, vc);
      // return result to client
      return rval;
    }

    // updates forwarded by other replica are added to local buffer
    receive Notify(u) {
      pending[origin(u.vc)].add(u);
    }

    // apply updates from buffer in causal order
    periodically for (sender: int) {
      if(forall i: (i=sender) || vc[i] ≥ pending[sender].next.vc[i]) {
        var u = pending[sender].dequeue();
        known.add(u);
        vc[sender] = u.vc[sender];
      }
    }
  }

}
```

References

[1] Sebastian Burckhardt. *Principles of Eventual Consistency.* Foundations and Trends in Programming Languages. now publishers, October 2014.

B Operational semantics for Antidote

Operational Semantics for Antidote

Peter Zeller
Deepthi Akkoorath
Annette Bieniusa

1 System State

There is a set of programs running concurrently and interacting with Antidote. We write p to refer to a single program.

We treat programs as abstract state transition systems. The implementation of a program p is given by a transition function f_p which takes the current local state and returns the action to perform. An action is one of the following:

localStep(ls') Perform a local step to state ls'.

beginAtomic$(ls'$, ***depTxn***$)$ Start a transaction with transaction $depTxn$ as causal dependency. The transaction snapshot is guaranteed to include $depTxn$.

endAtomic(ls') Commits the current transaction. The function ls' uses the identifier of the committed transaction to calculate the new local state.

dbOperation(ls', op) Calculates a database operation op to perform. The function ls' uses the result of the database call to calculate the new local state.

The configuration of a whole system (denoted by C in the following) consists of the local states of the programs together with the database state. We do not model the database state explicitly to allow for multiple implementations. Instead, we model the database state as the set of database-calls together with the happens-before relation on the calls. The concrete parts of configurations are listed below:

localState**(p)** The local state of program p

currentTransaction**(p)** The current transaction in program p

committedTransactions The set of committed transactions

calls**(c)** The information about database-call c. Is either $\bot$ or a pair (op, res) of operation and result. For database-updates, which do not have a result, we just write down the operation and omit the result.

The operation contains the key of the addressed object.

***callOrigin*(c)** The originating transaction for call c.

***visibleCalls*(t)** The set of calls visible to a transaction t

happensBefore The happens-before relation between database-calls

2 Operational Semantics

The rules of the operational semantics are given in Figure 1. Before giving an intuition about the rules, we explain the notation used in the rules:

- $C \xrightarrow{p,a} C'$ denotes that the system makes a step from configuration C to C' by executing a in program p.
- $C \xrightarrow{tr}{}^{*} C'$ denotes that the system makes zero or more steps from configuration C to C' by executing trace tr.
- The function *querySpec* specifies the result for database operations based on a given operation context. For example the *contains*-operation of an add-wins set s could be specified as:

  $$\begin{aligned} &querySpec(C, contains(s,x)) = \\ &\qquad \exists c_a.\ \mathit{calls}(C, c_a) \triangleq add(s,x) \\ &\qquad\quad \wedge \nexists c_r.\ \mathit{calls}(C, c_r) \triangleq remove(s,x) \wedge (c_a, c_r) \in \mathit{happensBefore}(C) \end{aligned}$$

- The operation context is a part of the state restricted to the set of visible calls in a transaction:

  $$operationContext(C,t) = \begin{bmatrix} \mathit{calls} = \mathit{calls}(C)|_{\mathit{visibleCalls}(t)} \\ \mathit{happensBefore} = \mathit{happensBefore}(C)|_{\mathit{visibleCalls}(t)} \end{bmatrix}$$

- $S \downarrow_R$ is the downwards-closure of set S with respect to relation R.
  $S \downarrow_R = \{x \mid \exists y \in S : (x,y) \in R^*\}$
- $x \triangleq y$ is short for $x = y \wedge y \neq \bot$

2.1 Intuition

It is always possible for one of the programs to perform a local step (rule *local*), which models local computations of the program without database interaction.

The three remaining rules for single-steps model the interaction of programs with the Antidote database. All database-operations are performed within a transaction in our model[1].

Transactions in Antidote work on a causally consistent snapshot. We model a snapshot as the set of database calls, which are visible in the given transaction. When a transaction

[1] Antidote also supports requests outside of interactive transactions, but those are merely performance optimizations and behave as if they were executed in their own transaction

$$\frac{\mathit{localState}(C,p) \triangleq \mathit{ls} \qquad f_p(\mathit{ls}) = \mathit{localStep}(\mathit{ls}')}{C \xrightarrow{p,\mathit{local}} C\left[\mathit{localState}(p) := \mathit{ls}'\right]}\;\textit{(local)}$$

$$\frac{\begin{array}{c}\mathit{localState}(C,p) \triangleq \mathit{ls} \qquad f_p(\mathit{ls}) = \mathit{beginAtomic}(\mathit{ls}', \mathit{depTxn}) \\ \mathit{currentTransaction}(C,p) = \bot \qquad t \text{ fresh} \qquad \mathit{txns} \subseteq \mathit{committedTransactions}(C) \\ \mathit{depTxn} = \bot \vee \mathit{depTxn} \in \mathit{txns} \qquad \mathit{vis} = \{c \mid \mathit{callOrigin}(C,c) \in \mathit{txns}\} \downarrow_{\mathit{happensBefore}(C)}\end{array}}{C \xrightarrow{p,\mathit{beginAtomic}} C \begin{bmatrix} \mathit{localState}(p) := \mathit{ls}' \\ \mathit{currentTransaction}(p) := t \\ \mathit{visibleCalls}(t) := \mathit{vis} \end{bmatrix}}\;\textit{(begin-atomic)}$$

$$\frac{\mathit{localState}(C,p) \triangleq \mathit{ls} \qquad f_p(\mathit{ls}) = \mathit{endAtomic}(\mathit{ls}') \qquad \mathit{currentTransaction}(C,p) \triangleq t}{C \xrightarrow{p,\mathit{endAtomic}} C \begin{bmatrix} \mathit{localState}(p) := \mathit{ls}'(t) \\ \mathit{currentTransaction}(p) := \bot \\ \mathit{committedTransactions} := \mathit{committedTransactions}(C) \cup \{t\} \end{bmatrix}}\;\textit{(end-atomic)}$$

$$\frac{\begin{array}{c}\mathit{localState}(C,p) \triangleq \mathit{ls} \qquad f_p(\mathit{ls}) = \mathit{dbOperation}(\mathit{ls}', \mathit{op}) \qquad \mathit{currentTransaction}(C,p) \triangleq t \\ c \text{ fresh} \qquad \mathit{res} = \mathit{querySpec}(\mathit{operationContext}(C,t), \mathit{op}) \qquad \mathit{visibleCalls}(C,t) \triangleq \mathit{vis}\end{array}}{C \xrightarrow{p,\mathit{dbOp}} C \begin{bmatrix} \mathit{localState}(p) := \mathit{ls}'(\mathit{res}) \\ \mathit{calls}(c) := (\mathit{op}, \mathit{res}) \\ \mathit{callOrigin}(c) := t \\ \mathit{visibleCalls}(t) := \mathit{vis} \cup \{c\} \\ \mathit{happensBefore} := \mathit{happensBefore}(C) \cup \mathit{vis} \times \{c\} \end{bmatrix}}\;\textit{(DB-operation)}$$

$$\frac{}{C \xrightarrow{\epsilon}{}^{*} C}\;\textit{(steps-empty)} \qquad \frac{C_1 \xrightarrow{tr}{}^{*} C_2 \qquad C_2 \xrightarrow{a} C_3}{C_1 \xrightarrow{tr \cdot a}{}^{*} C_3}\;\textit{(steps)}$$

Figure 1: Fine-grained interleaving semantics

is started (rule *begin-atomic*), we determine the transaction snapshot as follows: We first (nondeterministically) pick a set of already committed transactions (*txns*) for the snapshot. If the program provided a dependent transaction (*depTxn*), it is guaranteed to be included in this set of transactions. Then the set of visible calls for the snapshot is determined by taking all database calls which originated (*callOrigin*) in one of the chosen transactions (*txns*) and adding their causal dependencies ($\{\dots\} \downarrow_{\mathit{happensBefore}(C)}$).

The steps above provide some nice guarantees:

1. We only consider committed transactions, which provides a weak form of isolation. Concurrent transactions do not affect another running transaction. However, we do not have full isolation (as in serializability) as the snapshot might give a stale view on the database.

2. We can only pick complete transactions, which ensures that transactions are atomic: Either all calls or no calls of a transactions are visible.

3. The snapshot is guaranteed to be causally consistent as we include all causal dependencies based on the happens-before order.

4. By giving a dependent transaction, the program can influence session-guarantees. If the program provides the last transaction it has performed, it is guaranteed that the program observes its previous writes, that reads are monotonic, and that operations are performed in causal order (i.e. Read Your Writes, Monotonic Reads, Writes Follow Reads, and Monotonic Writes are all guaranteed).

When a transaction is committed (rule *end-atomic*), we simply add the transaction to the set of committed transactions. The transaction-identifier is returned to the program and can be used by the program to determine the next local state.

Executing an operation inside a transaction (rule *DB-operation*) is handled as follows: We first extract the operation context from the current state. The operation context consists of the database calls and the happens-before relation restricted, where both are restricted to the set of currently visible calls in the transaction. We then pass this context and the database-operation to the *querySpec*-function, which yields the result of the database operation.

Intuitively, the *querySpec*-function is a CRDT-specification. As the function only depends on the context and the operation, it is clear that there must be a CRDT implementation for every computable *querySpec*-function: The naive implementation simply records the history and applies the specification function. The reverse is also true[2]: Every CRDT implementation can only depend on the operations it has observed and since CRDTs are convergent there must be a deterministic function describing the result of queries, independent of the order in which operations where applied, only the happens-before order is relevant.

When an operation is executed, the result can be used by the program to determine its next local state. Additionally we add the call to the visible calls of the current transaction and record the operation in the history by updating *calls*, *callOrigin*, and *happensBefore*. The happens-before relation then includes an edge from all currently visible calls to the new call. This captures potential causality.

[2]If we ignore CRDTs which use additional information like timestamps. To model those, the semantics described here could easily be extended.

C References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by delta-mutation. In *Proc. of NETYS'15*, Morocco, 2015.

[2] Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of highly-available eventually-consistent data stores. *IEEE Trans. on Parallel and Dist. Sys. (TPDS)*, 28(1):141–155, January 2017.

[3] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *Symp. on Reliable Dist. Sys. (SRDS)*, pages 31–36, Montréal, Canada, September 2015. IEEE Comp. Society, IEEE Comp. Society.

[4] Sebastian Burckhardt. *Principles of Eventual Consistency*, volume 1 of *Foundations and Trends in Programming Languages*. Now Publishers, October 2014.

[5] Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *Int. Symp. on Principles and Practice of Declarative Programming*, pages 184–195, Siena, Italy, 2015. ACM.

[6] Marc Shapiro, Annette Bieniusa, Nuno Preguiça, Valter Balegas, and Christopher Meiklejohn. Just-Right Consistency: reconciling availability and safety. Rapport de Recherche 9145, Inria Paris; Sorbonne Universités; Tech. U. Kaiserslautern; U. Nova de Lisboa; U. Catholique de Louvain, Paris, France, January 2018.

[7] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[8] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm Strong Enough: Reasoning about consistency choices in distributed systems. In *Symp. on Principles of Prog. Lang. (POPL)*, pages 371–384, St. Petersburg, FL, USA, 2016.

[9] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching internet services with peer-to-peer interactions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 283–292, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.

[10] Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-crdts: Making δ-crdts delta-based. In *Proc. of the PaPoC'16 Workshop*, United Kingdom, 2016. ACM.