LIGHTKONE

Lightweight computation for networks at the edge

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

D4.2: Extended programming model for edge computing

Deliverable no.:	D4.2
Title:	Extended programming model for edge computing
Due date of deliverable:	January 15, 2019
Actual submission date:	January 14, 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
10/12/2018	1.1	File creation	UCL
10/12/2018	1.2	Initial structure	UCL
06/01/2019	1.3	First complete structure	UCL
10/01/2019	1.4	First complete content	UCL
14/01/2019	1.5	First complete deliverable	UCL
14/01/2019	2.0	Complete deliverable	UCL

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

Contributors:

Contributor	Institution
Peter Van Roy	UCL
Marc Shapiro	Sorbonne-Université
Nuno Preguiça	NOVA ID

Contents

1 Executive summary

Main progress The main progress of the second period (Month 13 - Month 18) is given by the following items:

- *Extended programming model*. The extended programming model adds abilities needed for practical applications, in particular the use case scenarios defined by the industrial partners. These abilities include for Antidote an improved SQL interface (called AQL), for Lasp, a port to the GRiSP embedded system boards together with a task model that allows defining computations on networks of these boards, and for Legion, an improvement of scalability of the peer-to-peer architecture, and support for authorization and programming abstractions to reduce user-perceived delays.

- *Just-right consistency JRC)*. The JRC methodology supports the development of heavy edge applications written using Antidote. In the second period, we worked on tool support for JRC, in particular the CEC tool (Correct Eventual Consistency), which has been released as a software artifact. The CEC tool aids the developer of highly available distributed applications by evaluating the correctness according to the JRC methodology (which is based on the CISE logic developed earlier). If the program is incorrect, the CEC tool provides counterexamples to show incorrectness and provides suggestions to the developer on how to fix the problem.

Main innovations with respect to state of the art During the second period, the core programming innovation of LightKone with respect to existing edge architectures, namely the convergent store, was strengthened in four ways. We added a task model to Lasp and an SQL interface to Antidote, which both strengthen the computation aspect of this innovation. We increased scalability of Legion and added the ability to reify CRDT update operations to reduce perceived user latency, which both strengthen the consistency aspect of this innovation.

Exploratory work In addition to the LiRA innovations mentioned above, we have done the following exploratory work. First, we explore extensions and variations on the CRDT concept in the context of a global-local view and referential integrity. Second, we explore the future of edge computing: migration and autonomy, and how to build and verify distributed systems. Third, we present an interesting new use case, namely the Red Wedding problem.

- *Global-local view*. This work defines a new model for shared objects, in which objects have a local view (per thread) and a global view. Operations on the local view are fast, and the model provides synchronization operations for the global view. This approach is an alternative way to define shared objects, as compared to CRDTs, and allows us to better explore the space of CRDT variants for LightKone.

- *Ensuring referential integrity under (only) causal consistency*. Referential integrity (RI) of a distributed object store means that all references inside of structured data are valid, both locally and in a distributed setting. We define a new CRDT, called *reference CRDT*, that allows to enforce RI with only causal consistency and liveness.

- *Rethinking distributed programming*. We reflect on the future of edge computing: how migration and autonomy will become more important, and how to build systems so that optimization and verification become much easier (by thinking about the use of declarative programming and about container interfaces). Finally, we present a new, interesting use case for edge computing, namely the Red Wedding problem, which has the particularity that it requires many stateful computations directly at the edge.

2 Introduction

The present deliverable D4.2 documents the progress made on programming models in LightKone during the first six months of 2018 (Month 13 - Month 18). This document is a supplement to Deliverable D4.1. While we have made an effort to make this document self-contained, we do not repeat all the information of D4.1 and we recommend that D4.1 be read completely before reading D4.2.

Since the rewritten versions of D4.1 and D4.2 are submitted simultaneously, for clarity we have consolidated in Deliverable D4.1 the main presentation of the three-year plan on programming models (see Section 3.1) and the main presentation of the state of the art comparison (see Section 5).

This document is structured as follows:

- Progress and plan (Section 3). This section gives the incremental progress for Month 13 - Month 18, basically in two areas: the extended programming model and just-right consistency (JRC).
- Software (Section 4). This section gives the links how to access the software deliverables and their documentation. The links for Lasp, Antidote, and Legion have remained the same, however we have added links for the CEC tool (Correct Eventual Consistency), which supports JRC.
- State of the art (Section 5). This section gives in the incremental update to the state of the art comparison given in Deliverable D4.1.
- 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 during the period 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.
- Programming models and runtimes (Appendix A). This appendix gives a self-contained presentation of the extended programming model as of Month 18, as it appears in an invited chapter in a book.
- References (Appendix B). 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.

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 progress of LiRA in the second period is explained in the present document.
- Explanation of the need to define a new semantics is given in Deliverable D4.1. As explained there, the LiRA programming model itself is an innovation that does not exist in any of the major edge programming 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 impact and status of the programming model work in the adopted platforms is explained in Section 3.1.
- The achieved results of work package 4 during the second 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 D4.2, 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

3.1 Plan

Deliverable D4.1 has presented the three-year plan for work on programming models in the project. Here we give an update to this plan at Month 18, with six months of additional work. The main plan during this period is to target the programming model work toward the needs of practical applications, and specifically the use cases. This is made clear by the explanations in Section (a), below, regarding the Antidote functionality added (for Scality and Guifi use cases), the Lasp port to the GRiSP boards and its extension with a task model (for Gluk use case), and the Legion work to increase scalability and improve user interaction quality (for mobile applications). These extensions were made in accord with LiRA and its programming model semantics.

(a) Impact and success measure

A coherent programming model is necessary because the chief innovation of LiRA, namely a convergent data store implemented in Antidote, Lasp, and Legion, combines several aspects usually handled separately by developers, namely storage, communication, computation, and consistency. The only way to have confidence that these aspects work well together is to define a unified semantics.

The impact of the unified semantics is difficult to quantify, since its success is marked by a *lack* of problems that would appear if it would not exist. For example, if the store would not have a clear consistency semantics, then fault injection would give erratic behavior, where sometimes the system would behave incorrectly due to inconsistent data between nodes.

However, it is possible to give a qualitative success measure for the semantics: Success is achieved if the erratic behavior is *not* observed, or if observed, it is *easily corrected*

since it is due to an error in how the semantics is implemented. Lack of a semantics leads to situations where bugs are corrected locally with no global coherence. The semantics ensures that bug corrections are globally coherent. Success is achieved if this coherence is achieved.

(b) Status of the programming model work

It would be a significant effort to define a unified semantics for all LiRA components. We do not have the resources for this in work package 4, but we have sufficient resources to solve the essential problems. In the case of LightKone, the first essential problem was to unify Lasp and Antidote, because these two components represent extremes in the spectrum of edge computing (namely light edge, at the extreme edge with no cloud connectivity, and heavy edge, with implicit cloud connectivity). We succeeded in defining a simple semantics that covers both (given in Deliverable D4.1). Achieving this simplicity took significant effort, but once achieved, the simplicity will continue to pay dividends for the rest of the project.

Solving the problem for these two extremes goes far to ensure that any problems encountered in intermediate cases will be much easier to solve. This is why, given the limited resources of work package 4, we targeted first the unification of the extreme cases. In general, it is important to solve difficult problems first, since if they are not solved, the project's evolution can make them even more difficult to solve in the future, until the project reaches a crisis state where the goals cannot be achieved. We consider that the achievement of the unified semantics for Lasp and Antidote is a necessary condition to ensure the project will not fail because of incompatible component behavior. To attach a percentage figure to this achievement, it is important to remember how we prioritize difficult problems, so that they are solved early. In terms of conceptual breakthroughs, we can consider that we have achieved more than 50% of the results required for LightKone success, even if we have spent less than 50% of the person-months. This gives us the resources to help ensure the success of the rest of the project. In the third year, we will focus on managing the connection between the semantics and the software artifacts, to ensure that the coherence is maintained as the artifacts evolve and are used to implement the use case scenarios. This will likely require extensions and modifications to the semantics, but this is acceptable since the semantics will at all times exist, and its existence is what guarantees coherence.

3.2 Progress

With respect to the first period, two main areas of progress were made with respect to D4.1, namely for the extended programming model and for just-right consistency.

(a) Extended programming model

The extended programming model is an improved version of the basic programming model released in Jan. 2018. Improvements were made in Antidote, Lasp, and Legion:

- For Antidote, the AQL interface was improved to increase coverage of SQL, in particular for the *select*, *update*, and *delete* statemenets. The implementation work

needed to support this, in particular the support for secondary indexes, is further explained in Deliverable D6.2.

- For Lasp, the system was ported to GRiSP embedded system boards and a task model was added to permit storing programs inside the Lasp storage. The implementation details of this support are explained in Deliverable D5.2.

- For Legion, scalability was improved in the peer-to-peer architecture, security was added (to prevent unauthorized actions), and programming abstractions were added to reduce perceived user delay. This is explained further in Deliverable D5.2.

Section 3.3 gives more information on these improvements.

(b) Just-right consistency (JRC)

Just-right consistency is a programming methodology for Antidote, to maintain application invariants despite concurrency and partitioning problems. This was introduced in Deliverable D4.1 and is explained there. Work on tool support for JRC continued in the second year. There is continued work on the CEC tool (Correct Eventual Consistency), to provide suggestions to the user how to fix the program when verification fails. We started work on a second tool, called Soteria, to verify that the state update for CRDTs satisfies the monotonicity properties for correct operation. Section 3.4 gives more information on these improvements.

3.3 Extended programming model

The extended programming model was described in an invited chapter in an upcoming book on Ultrascale Computing Systems, which is being organized by the NESUS project. The chapter focuses specifically on General Purpose Computations at the Edge and is given in Appendix A. This text gives a presentation of the work written concisely and clearly for a general audience, so we include it as part of the present document. This text gives a snapshot as of April 2018 of the implemented LightKone technology for edge computing, including synchronization-free computing with CRDTs and hybrid gossip. However, it is written in order to be up-to-date with the progress during the first six months of 2018.

3.4 Just-right consistency (JRC)

To help the developer apply Just-Right Consistency, we are developing tool support for formally verifying applications. D4.1 and D6.1 explain the beginning of this work on tools, namely the CISE proof system, to prove that the application satisfies a specification in first-order logic. We present here the progress of this work as of Month 18, for the CEC tool (Correct Eventual Consistency) and a new tool called Soteria.

(a) Correct Eventual Consistency Tool

Reasoning about the correctness of distributed applications while ensuring high availability is a non-trivial task. The CEC (Correct Eventual Consistency) tool described in D6.1,

is geared towards helping the developers reason about the correctness of their designs. The tool is based on CISE logic developed by Gotsman et. al. [5].

The technical report [6], mentioned in D6.1, was based on evaluating the tool. The report highlighted the lack of proper debugging support when verification condition fails. The tool has been evolved to generate counter examples which are comprehensible to the user. The tool helps the user identify the statement which failed, and provides the values of the variables in that statement for the failed verification condition. It also provides values for the parameters of the procedure call. These two informations can help developers identify the issue with their specification.

As a next step, the counter example obtained from the previous step is used to provide token suggestions to the user. Currently we are looking into providing inequality suggestions, suggesting which parameters if not equal can help ensuring correct execution. The previous method of token suggestion was using a brute force approach, where in all pair-wise combinations of the parameters involved were considered and tests were run for all of them. In the new approach, we utilise the counter example and sees the parameters which share the same value. We then enforce an inequality constraint on that particular pair of parameters and rerun the verification condition with those restrictions. If the verification succeeds, we suggest them to the user. So far our results are matching the brute force approach employed earlier, but taking less time since we do not run through all possible combinations.

Consider an example of a bank account having just two simple operations - deposit and withdraw. The tool runs the three checks explained in D6.1. The third check, the stability check for withdraw(account_id, amount) and withdraw(account_id, amount) will fail and produce a counter example indicating the values of two account_id's being equal. That means the withdraw operation does not preserve its precondition and it needs some concurrency control token. So the tool enforces an inequality constraint between the account_id's and checks whether the verification succeeds. In this case, it is a yes and so the tool adds it to the list of token suggestions. For a developer that means the withdraw operations operating on one account_id need to synchronize.

The run-time of the tool is available in a github repository given in Section 4.4. Some examples are provided in the folder specifications/applications. There is a readme in the repository which talks about how to use the tool and how to write specifications.

(b) Soteria

The previous tool talks about the design of distributed applications when the effects of operations are propagated. Another design in popular use in distributed applications is when the whole state is propagated to other replicas. There are currently no tools available to enable the developer to reason about those types of applications. To address this issue of state-based update propagation, we are working on building a tool named Soteria.

For ensuring convergence in state-based update propagation scenario, we rely on state-based CRDTs [1]. State-based CRDTs ensue convergence if all the operations are monotonically non-decreasing and there is a least upper bound for any two states. The developer needs to define two operations:

- the merge function, which specifies how to merge the incoming state with the current state in the replica; and

- the comparison function to check the properties for a semi-lattice.

To check whether the application operations and merge conform to a semi-lattice, the tool does the following checks:

- whether each update is monotonically non-decreasing;
- whether the merge function provides the least upper bound of two states;
- whether each update and merge upholds the invariant during a sequential operation; and
- whether the stability of the precondition of merge (precondition of merge is essentially the set of reachable states) under each update and merge itself.

The specification input for the tool is given in Boogie [2].

4 Software

The extended programming model is released as three software artifacts, namely Lasp, AntidoteDB, and Legion. The software has been improved since Month 12, but is released in the same Web locations.

We add a new software artifact, the CEC tool, which supports the Just-Right Consistency methodology.

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

4.4 CEC tool (Correct Eventual Consistency)

Documentation See the folder specifications/applications in the code repository.

Code repository https://github.com/LightKone/correct-eventual-consistency-tool

5 State of the art

The main presentation of the state of the art in programming models for edge computing was given in D4.1, and the main LightKone innovations with respect to state of the art architecture are presented in that deliverable. In the present section, we explain the increment to these innovations due to the progress made in the second period. This progress has strengthened the main innovation of LightKone, namely the convergent data store, in the following ways:

- The Lasp programming model was extended with a task model, allowing to do computations inside the Lasp system. This increases expressivity of the computation aspect of the convergent data store.
- The Antidote programming model was extended with an SQL interface. This increases expressivity of the computation aspect of the convergent data store.
- The Legion programming model was extended to increase scalability, to add authorization, and to reduce perceived use delay by reifying the CRDT update operations. This strengthens the consistency aspect of the convergent data store.
- The CEC tool in the JRC methodology increases the usefulness of this methodology by giving advice to developers on correcting errors. This strengthens the consistency aspect of the convergent data store.

6 Exploratory work

We give the exploratory work done from Month 13 to Month 18.

6.1 Global-local view: scalable consistency for concurrent data types

We propose a new model for shared objects which leverages the different views of an object, called the *global-local view model* (published in EuroPar 2018, see Section 7.1). In this model, each thread has a local view of the object which is isolated from other threads. Threads update and read the local view. The local updates, though visible in a local view, are made visible on a global view only after an explicit two-way merge operation is performed. The other threads observe these changes once they synchronize their local view with the global view by the merge operation. As the local view is non-shared, the local updates can be executed without requiring synchronization, thus enabling better performance, albeit at the expense of linearizability. We discuss the design of several datatypes and evaluate their performance and scalability compared to linearizable implementations.

In addition to the local operations, the model also provides synchronous operations on the global view. Consider, for example, a queue where the enqueues have been executed on the local view. To guarantee that the elements are dequeued only once, dequeues are executed atomically on the global view. We call the operations that perform only on local view, weak operations and those on the global view, strong operations. Combining operations on the global and the local views, we can build data types with customizable semantics on the spectrum between sequential and purely mergeable data types. Mergeable data types provide only weak and merge operations; hybrid mergeable data types

offer both weak and strong operations. An application that uses a hybrid mergeable data type may use weak updates when a non-linearizable access is sufficient and can switch to use only strong operations when stronger guarantees are required.

6.2 Ensuring referential integrity under (only) causal consistency

Referential integrity (RI) is an important correctness property of a shared, distributed object storage system. It means that all references in structured data are valid, both locally and in a distributed setting. Updating a distributed database while maintaining RI is nontrivial. It is sometimes thought that enforcing RI in a distributed system requires a strong form of consistency. We argue that causal consistency with liveness suffices to maintain RI. We support this argument with pseudocode for a *reference* CRDT data type that maintains RI under causal consistency. QuickCheck has not found any errors in the model. This work was published and presented at the PaPoC workshop [7]. The work is important to LightKone because it makes a connection between CRDTs and referential integrity, which may be important for future extensions to our CRDT storage.

(a) Defining the RI problem

Consider a shared store (memory) of objects, and a *reference* data type for linking objects in the store. Intuitively, the *referential integrity* (RI) invariant states that if some source object contains a reference to some target, then the target "exists," in the sense that the application can access the target safely. A referenced object must not be deleted; conversely, when an object cannot be reached by any reference, deleting it is allowed.

We say that an object is *unreachable* if it *is not* the target of a reference, and *never will be* in the future (the latter clause is problematic under weak consistency). The RI property that we wish to achieve is the following:

- *Safety:* An object can be deleted only if it is unreachable.
- *Liveness:* Unreachability of an object will eventually be detected.

In a storage system where the application can delete objects explicitly, the programmer must be careful to preserve the RI invariant. Our paper exhibits a reference data type demonstrating that causal consistency (with progress guarantees) suffices to ensure RI and to implement a safe deletion operation. The solution uses a form of reference counting (designed for distributed systems), called *reference listing* [3, 4, 8]. Objects with a non-empty reference list must not be deleted.

(b) Sketch of solution

A source object contains an instance of a data type called `outref` for every attribute that refers to another object. A (target) object is associated with exactly one `inref`. The inref identifies the currently-known sources pointing to this target. The only application-level operations supported by inref are initialisation and testing whether deleting the target is allowed.

A (source) object contains any number of distinct outrefs. An outref supports the following application-level operations: *(i)* initialisation, *(ii)* assigning from another outref, *(iii)* assigning **null** (we assume that deleting an object first automatically nulls out all of its outrefs), and *(iv)* invoking its target(s). To support concurrency, assigning an outref behaves much like a *Multi-Value Register* [9]. Assignment overwrites its previous

value; when concurrent assignments occur, the resulting reconciled value contains all the concurrently-assigned values.

Our algorithm design hinges on two principles that can be implemented assuming only causal consistency: (1) *before* an outref is assigned to a source object (in initialisation or assignment), we ensure that the corresponding inref has been added to the target object; importantly, causal consistency is enough to enforce this ordering of updates. (2) To delete a target, we require that no inref exists, nor will later be added, for this target. This property can be checked by well-known mechanisms which rely only on causal consistency and progress guarantees [10]. The combination of these properties is sufficient to ensure RI as defined above. For more detail, we refer to the paper [7].

6.3 Rethinking distributed programming (continued)

This section continues the same-named section from Deliverable D4.1 with five additional exploratory papers. These papers investigate different aspects of edge computing: two papers that explore future architectures for edge computing, a paper on how to use declarative programming for distributed computing, a paper on verification for container-based components, and a paper on an interesting new use case for edge computing, namely the Red Wedding problem.

(a) Towards enabling novel edge-enabled applications

This paper gives a perspective on the future of edge computing, in which applications can move freely between edge and cloud. The paper identifies the key research challenges that this implies: decentralized scalable resource management, migration, replication and decomposition of computational tasks, partial replication, lightweight and scalable monitoring, autonomic management for resources and workloads, and cryptography for data privacy and integrity. This paper is published as a technical report (see Section 8.3).

(b) A case for autonomic microservices in the edge

This paper is a companion to the above paper and argues that edge computing will lead to a much greater need for autonomic services that manage computational and data resources. An important requirement is elastic monitoring, which itself becomes part of the edge infrastructure. This paper is submitted to SOCC 2018 (see Section 8.2).

(c) A software system should be declarative except where it interacts with the real world

This paper is part of an ongoing effort to understand precisely the concept of *synchronization-free*, as used in the SyncFree and LightKone projects. As part of this effort, we determine a design principle for software systems that lets us understand where declarative programming paradigms (such as logic and functional programming) should be used. Basically, declarative programming should be used everywhere except with real-world interactions, which are defined as interactions in which wall-clock time is important. This paper is published in LPOP 2018 (see Section 7.2).

(d) Verifying interfaces between container-based components

Container-based programming has emerged as a basic technology for building distributed applications, and is therefore important for edge applications. An important requirement is the ability to compose black-box components, such that the composition preserves component invariants. One way to make progress in this area is to leverage work on type systems, because type systems are already doing partial verification of some invariants. This paper is submitted to HotEdge 2018 (see Section 8.2).

(e) Towards a solution to the Red Wedding problem

An important real-world scenario for edge applications is stateful computations at the edge. Spikes in stateful edge computations occur because of real-world events, such as the Red Wedding episode of Game of Thrones, which caused many people to simultaneously update information stored at the edge. A realistic edge platform has to be able to handle such spikes and this paper proposes a design for such a platform. This paper is published in HotEdge 2018 (see Section 7.2).

7 Published papers

7.1 Refereed conference papers

- Deepthi Devaki Akkoorath, José Brandão, Annette Bieniusa, and Carlos Baquero. *Global-Local View: Scalable Consistency for Concurrent Data Types*. 24th International European Conference on Parallel and Distributed Computing (Euro-Par 2018), Aug. 27-31, 2018 *(to appear)*.

7.2 Refereed workshop papers

- Peter Van Roy. *A Software System Should be Declarative Except Where it Interacts with the Real World*. Workshop on Logic and Practice of Programming (LPOP 2018, colocated with FLoC 2018), Oxford, UK, July 18, 2018 *(to appear)*.

- Christopher S. Meiklejohn, Heather Miller, and Zeeshan Lakhani. *Towards a Solution to the Red Wedding Problem*. USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18), July 10, 2018 *(to appear)*.

- Marc Shapiro, Annette Bieniusa, Peter Zeller, and Gustavo Petri. *Ensuring referential integrity under causal consistency*. Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data (PaPoC 2018, colocated with EuroSys 2018), Porto, Portugal, April 23-26, 2018.

7.3 Book chapters

- Ali Shoker, João Leitão, Peter Van Roy, and Albert van der Linde. *Programming Models and Runtimes: Towards General Purpose Computations at the Edge*. The IET. Book chapter. Ultrascale Computing Systems. 2018 *(in press)*.

8 Other dissemination

8.1 Invited talks

- Marc Shapiro. *Just-Right Consistency: As available as possible, as consistent as necessary, correct by design*. Invited talk. dotScale Paris, June 2018, Paris, France.

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

- Annette Bieniusa. *Just the right kind of Consistency!*. Keynote talk. Typelevel Summit Berlin, May 18, 2018, Berlin, Germany. See https://typelevel.org/event/2018-05-summit-berlin/.

- Marc Shapiro. *Just-Right Consistency: As available as possible, consistent when necessary*. Invited talk. Workshop on Verification of Distributed Systems (colocated with NETYS 2018), May 6-8, 2018, Essaouira, Morocco.

- Carla Ferreira. *Verification tools for available and correct distributed applications*. Invited talk. Workshop on Verification of Distributed Systems (VDS 2018), Essaouira, Morocco. May 6-8, 2018, Essaouira, Morocco. See http://netys.net/VDS2018.html.

- Annette Bieniusa. *Highly-Available Applications Done Correct*. Dagstuhl Seminar 18091, Feb. 25 -– Mar. 2, 2018, Wadern, Germany.

- Marc Shapiro. *Just-Right Consistency: As available as possible, as consistent as necessary, correct by design*. Dagstuhl Seminar 18091, Feb. 25 -– Mar. 2, 2018, Wadern, Germany.

- Peter Van Roy. *Elements of a Unified Semantics for Synchronization-free Programming Based on Lasp and Antidote*. Dagstuhl Seminar 18091, Feb 25 -– Mar 2, 2018, Dagstuhl, Germany.

8.2 Submitted papers

- João Leitão, Maria Cecília Gomes, Nuno Preguiça, Pedro Ákos Costa, Vitor Duarte, David Mealha, André Carrusca, and André Lameirinhas. *A Case for Autonomic Microservices for Hybrid Cloud/Edge Applications*. Submitted to SOCC 2018 as a *Vision Paper*.

- Christopher S. Meiklejohn, Zeeshan Lakhani, Peter Alvaro, and Heather Miller. *Verifying Interfaces Between Container-Based Components*. Submitted to HotEdge 2018.

8.3 Research reports

- João Leitão, Pedro Ákos Costa, Maria Cecília Gomes, and Nuno Preguiça. *Towards Enabling Novel Edge-Enabled Applications*. Technical Report, UNL, Lisboa, May 2018.

- Marc Shapiro, Annette Bieniusa, Nuno Preguiça, Valter Balegas, and Christopher Meiklejohn. *Just-Right Consistency: Reconciling Availability and Safety*, Rapport de Recherche INRIA 9145, Jan. 2018.

A Programming Models and Runtimes (NESUS book section)

This appendix gives a self-contained presentation of the LightKone programming model work, written for a general audience. This text is part of a chapter on General Purpose Edge Computing in a book on Ultrascale Computing organized by the NESUS project.

Chapter 1

Programming Models and Runtimes

Ali Shoker[1], João Leitão[2], Peter Van Roy[3], and Albert van der Linde[4]

1.1 Towards General Purpose Computations at the Edge

Originally designed to exploit the power of multi-core processors through virtualization, Cloud Computing [1] has changed over the past decade to support ultrascale computations. The new paradigm, often called *aggregation*, collects a large number of resources in a pool to form a single service with huge storage and computation capacities. Unfortunately, with the huge amounts of data generated via modern applications, the cloud center has become a bottleneck and a single point of failure. This advocated an extended paradigm, called *Edge Computing*, that brings part of the data storage and computation closer to the user. The benefits are plenty: reduced delays, high availability, low bandwidth usage, improved data privacy, etc. In this section, we introduce recent advances in edge computing that makes the coordination of edge networks synchronization-free and convergent. We address the main challenges facing applications on the data management and communication aspects. The section also provides convenient runtime environments for different categories of edge computing scenarios[5].

1.1.1 Motivation

Edge Computing offers the opportunity to build new and existing ultrascale applications that take advantage of a large and heterogeneous assortment of edge devices and environments. Fully realizing the opportunities that are created by edge computing, requires dealing with a set of key challenges related with the high number of different components that compose such systems and the interactions among them.

[1]HASLab, INESC TEC & University of Minho, Portugal
[2]Universidade Nova de Lisboa, Portugal.
[3]Université Catholique de Louvain, Belgium.
[4]Universidade Nova de Lisboa, Portugal.
[5]Credits go to all team members contributed to the success of this work within the EU FP7 Syncfree project and EU H2020 LightKone project. The research leading to these results has received funding from the European Union's Horizon 2020 - The EU Framework Programme for Research and Innovation 2014-2020, under grant agreement No. 732505, LightKone project.

In this work, was address the main challenges on the communication and data management levels allowing for robust communication and available data access.

On the communication frontend, the fact that applications are composed of components running in heterogeneous environments requires robust and efficient solutions for tracking these components. This implies the development of highly robust and adaptive membership services and mechanisms that allow efficient communication among these components. Among the promising class of gossip-based communication protocols are those "hybrid" ones [2, 3], in which payloads are propagated though an elected logical *spanning tree*, supported by lightweight meta-data across the graph for recovery (reconstructing another logical tree) under failures.

The consequences of such hostile environments are also present on the data management level. Since application components run on different administrative domains scattered across heterogeneous environments, communication links between these components can be disrupted by external factors (i.e, network partitions) frequently. This implies that the progress of computations executed across different application components cannot depend on continuous communication with other components, or in other words, cannot depend on synchronous interactions. This advocates the use of synchronization-free (i.e., sync-free) programming abstractions backed by sync-free data propagation and replication techniques. An interesting approach is to make use of Conflict-free Replicated Data Types (CRDTs) [4, 5, 6] that are proven abstractions designed to acheive convergence under such conditions (this is explained later in more details).

Finally, heterogeneity is the norm in ultrascale edge applications, and it exists at various layers: executiuon environments, communication media, data sources, operating systems, programming languages, etc. Addressing this heterogeneity can be achieved by leveraging on different run-time supports and frameworks that provide a more unified vision of resources to application developers. These different run-time and frameworks will have to inter-operate through the use of standard protocols and common data representation models.

In the following we refine the challenges associated with tapping on edge computing to design ultrascale applications, and discuss enabling technology that paves the way to tackle these challenges, and finally discuss a set of run-time and framework support that can simplify the design of such applications.

1.1.2 Edge Computing Opportunities

Edge Environments. To the contrary of cloud computing where the data and computation is centralized at the cloud data centers, the edge computing paradigm encompasses a large number of highly distinct execution environments that are defined by the network topology, connectivity, locality, and the storage and computation capacities of the devices used. In particular, we identify we identify the following interesting edge environments:

- Fog Computing: a variant of cloud computing where the cloud is divided into smaller cloud infrastructures located in the user vicinity. In such environments,

each fog cloud often serves as an individual cloud, although the data can eventually be incorporated with other fog clouds [7, 8].

- Mobile Cloudlets: small cloud datacenters that are located at the edge and are tailored to support mobile applications with powerful computations and low response times, e.g., in ISP gateways or 5G towers [9, 10, 11].
- Hardware-based Clouds: self-contained devices, such as routers, gateways, or set-up boxes, that are enriched with additional computational and storage capabilities like [12, 13].
- Peer-to-Peer (P2P) Clouds: these environments try to leverage existing devices, e.g., user mobiles, laptops, and computers in volunteer networks, aiming to cooperate towards achieving a common goal [14, 15, 2].
- Things and Sensor Network Clouds: resource constrained devices, e.g., Internet of Things devices, sensors, and actuators, capable of performing some computations on data without accessing or delegating to the (possibly unreachable) cloud center [16].

All of these different scenarios are characterized by having highly heterogeneous devices in terms of processing power and memory, but also regarding their connectivity to the backbone of the Internet or even their up-times (being continually running or being operating for only small periods of time). These different devices naturally, run different operating systems, from general purpose Linux based operating systems in the case of servers in cloud and private infrastructures, to proprietary operating systems in the case of set-up boxes, mobile operating systems, general purpose multi-user operating system or even single process operating systems in the case of small sensors and actuators. Gathering the capacity of devices with very different properties is highly challenging, and devising solutions that can exploit devices located in different edge devices brings additional challenges. Next we will discuss some of the key high level challenges in tapping the potential of the edge.

Challenges at the Edge Despite the diversity of edge computing environments, components, and properties, the major challenges are common to most of the scenarios. In particular, we recognize the following four challenges:

Scalability. One of the reasons to move the data and computation off the cloud data center to the edge is to reduce the I/O overload on the cloud and avoid bottlenecks related with the limited network capability connecting clients to the cloud infrastructures. Nevertheless, this raises another challenges on handling the data and computation in a distributed way especially in ultra-scale systems composed of, potentially, many data centers and thousands of edge devices. This scale requires special techniques across the data, computation, and communication planes. As captured by the CAP theorem [17], and because scaling out will increase the potential for network portions, link failures, and arbitrary communication delays, ensuring availability—as an essential requirement for most applications including novel edge applications—requires relaxing the consistency model employed in the design and implementation of these solutions. Consequently, the computation should also be

decentralized and coordinated to achieve the common goals of the entire system. Finally, the communication middlewares should also scale to afford a high number of nodes, e.g., through asynchronous, P2P, or gossip protocols.

Interoperability. Considering the edge categories discussed above, one can notice the notable diversity level of the devices and platforms used within the same or across edge clouds. This brings interoperability challenges if all components shall communicate with each others, thus requiring well studied interfaces and possibly introducing a common layer that all components can understand without compromising the characteristics deemed essential.

Resilience. While cloud datacenters use high quality equipment for the network and devices, edge computing often use commodity equipment that are far from perfect regarding failures. The problem is extrapolated with edge network problems that are likely to be loosely connected, mobile, and hostile. This threatens the quality of the service and makes the data and communication components even more complex. That said, one must consider the performance as well as the cost trade-offs (being a major factor due to the constrained resources).

Security and Privacy. Given the heterogeneity of the edge applications, security and privacy measures must be analyzed and tackled individually. However, in general, it is desired to find a common security layer or security measures that govern a wide range of applications. Security and privacy on the edge need to be addressed on the infrastructure and data levels. The former can be deployed at the communication or network layer, ranging from establishing secure connections to enforcing secure group dynamics, and cover several dimensions including data integrity, data privacy, or resilience to DoS attacks. On the other hand, edge applications often deal with sensitive data which likely requires lightweight encryption and data sanitization techniques to control the disclosure of such data. These may also include secret-sharing, anonymization, noise addition or partitioning, etc., depending on the specific security and functional requirements of the implementations.

Use Cases. As discussed in the edge environments, edge computing supports a plenty of applications and use-cases. In this section, we focus on three categories in which most of the use-cases lie:

- Time series applications. This category spans a multitude of applications with the popularity of IoT. The scenario is often a type of time series where data is generated by the IoT devices, e.g., sensors, and pushed to the edge devices to get stored, aggregated, and partially computed. The aggregated data is then pushed to the center of the cloud for further handling. The data-flow can sometimes be in the opposite sense if actuator devices exist; in this case, the processed data in the cloud is pushed back to the actuators to do some action. Consequently, this scenario represents a hybrid model of light and heavy devices, different types of networks (e.g., Zig-bee, WIFI, WAN, etc), as well as data-flow direction.
- Mobile edge applications. This category covers all the applications in which devices are mobile and public. This makes the model very hostile as link failure and delays are expected, and the availability of nodes cannot be guaranteed (e.g.,

a mobile device can be switched off). The communication in such use cases does not follow a particular data-flow pattern, but it is often P2P or gossip-based due to the dominant dynamic graph-like network of nodes. In such applications, devices have moderate storage and computation resources that makes the interaction symmetric. Obviously, the main challenges in such use-cases are reseliance and availability. In some cases, access points, towers, or routers with more capacities can assist in storage, computation, and communication, which can be used as third party authority when needed.

- Highly available databases. This category is a natural evolution of scalable databases in cloud and cluster systems. The intuition is to replicate the database geographically, brining replicas or cache servers closer to the user. In this scenario, devices are at least commodity computers or servers with non-scarce capacities, and then network is often the Internet. In addition to availability, the challenge in such use-cases is to tolerate network partitions and optimize data locality (especially when partial replication is used). These scenarios are close to Fog Computing and Cloudlets with the difference that all node must work as a single (often loosely) coordinated system.

1.1.3 Enabling Technologies for the Edge

Synchronization-Free Computing. Edge devices and edge networks are both unreliable. This follows both from their design, e.g., they are low-power systems that are often offline, and from the nature of the edge itself, e.g., it is directly involved with real world activities, such as in Internet of Things. Despite this unreliability, we would like to perform computations directly on the edge.

To perform computations directly on the edge, we need distributed data structures and operations that tolerate the unreliability of the edge. Synchronization-free computing fits the bill because of its very weak synchronization requirement. A prominent example is Conflict-free Replicated DataType (CRDT), which is a replicated data type that is designed to support temporary divergence at each replica, while guaranteeing that when all updates are delivered to all replicas of a given instance, they will converge to the same state. (More details about CRDTs can be found in Chapter 4 or by referring to [4, 5, 6].) CRDTs naturally tolerate node problems, namely nodes going offline and online and node crashes, and network problems, namely partitions, message loss, message reordering, and message duplication. Node crashes are tolerated as long as the desired state exists on at least one correct node. The following results on CRDT computations are summarized from [18].

CRDT Definition. For the purposes of this section, we define a *CRDT instance* to be a replicated object that satisfies the following conditions:

- Basic structure: It consists of n replicas where each replica has an initial state, a current state, and two methods, query and update, that each executes at a single replica.
- Eventual delivery: An update delivered at some correct replica is eventually delivered at all correct replicas.
- Termination: All method executions terminate.

- Strong Eventual Consistency (SEC): All correct replicas that have delivered the same updates have equal state.

This definition is slightly more general than the one given in the original report on CRDTs [4]. In that report, an additional condition is added: that each replica will always eventually send its state to each other replica, where it is merged using a join operation. This condition is too strong for CRDT composition, since it no longer holds for a system containing more than one CRDT instance. We explain the conditions needed for CRDT composition in the next section.

CRDT Composition. The properties of CRDTs make them desirable for computation in distributed systems. It is possible to extend these properties to full programs where the nodes are CRDTs and the edges are monotonic functions. To achieve this, it is sufficient to add the following two conditions on the merge schedule, i.e., the sequence of allowed replica-to-replica communications:

- Weak synchronization: For any execution of a CRDT instance, it is always true that eventually every replica will successfully send a message to each other replica.
- Determinism: Given two executions of a CRDT instance with the same set of updates but a different merge schedule, then replicas that have delivered the same updates in the two executions have equal state.

The first condition allows each CRDT instance to send the merge messages it requires to satisfy the CRDT conditions. The second condition ensures that the execution of each CRDT instance is deterministic, which makes it a form of functional programming. We remark that SEC by itself is not enough for this, since the states of replicas *in different executions* that have delivered the same updates can be different, even though SEC guarantees that they are equal in the same execution. In practice, enforcing determinism is not difficult but it depends on the type of the CRDT instance. Article [18] explains how to do it for a set that has add and remove operations (the so-called Observed-Remove Set).

We define a *CRDT composition* to be a directed acyclic graph where each node is a CRDT instance, and each node with at least one incoming edge is associated to a function of all incoming edges arranged in a particular order. Given the first of the two conditions introduced above, we can show that the execution of a CRDT composition satisfies the same properties as a single CRDT instance. If the second condition is added, then the CRDT composition behaves like a functional program.

Hybrid Gossip Communication. Gossip is a well known and effective approach for implementing robust and efficient communication strategies on highly dynamic and large-scale system [15, 3]. In its most simple form, in a gossip protocol, each node periodically interacts with a randomly selected node. In this interaction both exchange information about their local state (and potentially merge it). Since all nodes do this in parallel and in an independent fashion, after approximately one round-trip time, all nodes will have performed, at least, one merge step, and on average two merge steps (one initiated by the node itself and another initiated by some

peer). We usually call this period of interactions a *cycle*. After a small number of cycles, the network converges to a globally consistent vision of the system state. This simple approach cab be used, for instance to compute aggregate functions, such as inferring the network size or load. Interestingly, this can also be used for other, and more complex, purposes such as managing the membership of large-scale system, which implies building and maintaining an overlay (i.e, logical) network topology, in a way that is both robust and scalable, but also to support robust data dissemination in such systems.

Gossip-based approaches have been shown to be highly resilient to network faults, due to the inherent redundancy that its core to the design of gossip protocols. Unfortunately, this redundancy also leads to efficiency penalties. Hybrid gossip addresses this aspect of gossip protocols. In a nutshell, the key idea of hybrid gossip is to leverage on the feedback produced by previous gossip interactions among nodes, such that an effective and non-redundant structure of communication can naturally emerge. The topology of this *emergent structure* depends on the computation being performed by nodes, and it enables nodes significantly improve the communication and coordination cost by restricting the exchange of information among node to the logical links that belong to this structure, lowering the among of redundant communication.

Key to maintaining the fault-tolerance of gossip protocols in hybrid gossip is the use of the remaining communication paths among nodes (those that are not selected to be part of the emergent structure) to convey minimal control information. This control information enables the system to detect (and recover) from failures that might affect the emergent structure. Moreover, in highly dynamic scenarios, the additional communication paths allow nodes to fall back to a pure gossip strategy, for instance, when there are a significant number of concurrent nodes crashes or network failures.

Interesting, hybrid gossip solutions naturally allow different components of the system to operate using either the emergent structure or a pure gossip approach simultaneously. Hence, components of the system that are in stable conditions (i.e, low membership dynamics and low failures) will operate resorting to the emergent structure, while components of the system that are subjected to high churn or network/node failure will fallback to use pure gossip while still being able to interoperate with the components using the emergent structure.

Therefore, hybrid gossip approaches enable applications to, effectively and transparently, benefit from the resilience of a pure gossip approach entwined with the efficiency of a gossip approach that leverages an emergent communication topology. The hybrid gossip approach has been introduced in [2, 19]. The Plumtree protocol in particular, shows how to build an efficient and robust spanning tree connecting large number of nodes to support reliable application-level broadcast. This solution is currently used in industry, for example, the Basho Riak database uses it to manage the underlying structure of its ring topology which is used to map data object keys into nodes (through consistent-hashing).

1.1.4 Runtime for Edge Scenarios

Above we have discussed enabling technologies that can be leveraged to build new and exciting edge applications in the ultrascale domain. Tapping into these enabling technologies can however, be a complex task for developers. Therefore, it becomes relevant to provide frameworks, tools, and other artifacts that exploit these technologies in a coherent way, providing high level abstractions to programmers that aim at developing their ultrascale edge applications. We now discuss some existing runtime support tools and frameworks that have been recently proposed to this end.

Antidote. Antidote is a geo-replicated key-value store, designed for providing strong guarantees to applications while exhibiting high availability, thus providing a good compromise in the consistency versus availability trade-off in the design of cloud databases. These proprties make Antidote a strong candidate as an edge database especially when edge nodes have non-scarce resources (e.g., commodity servers).

In particular, some cloud databases adopt a strong consistency model by enforcing a serialization in the execution of operation, leading to high latency and unavailability under failures and network partitions. Other databases adopt a weak consistency model where any replica can execute any operation, with updates being propagated asynchronously to other replicas. This approach leads to low latency and high availability even under network partition, but replicas can diverge. On the other hand, Antidote allows any operation to execute in any replica, but provides additional guarantees to the application as we explain next.

First, Antidote relies on CRDTs for guaranteeing that concurrent updates are merged in a deterministic way. Antidote provides a library of CRDTs with different concurrency semantics, including registers, counters, sets and maps. The applications programmer must select the most appropriate CRDT, considering its functionality and concurrency semantics (e.g., add-wins, remove-wins).

Second, Antidote enforces causal consistency, guaranteeing that whenever an update u may depend on update v, if a client observes update u he also observes update v. Applications can leverage this property to guarantee their correctness when the correctness depends on the order of updates, e.g., an update executed after changing the access control policies should not be visible in a replica with the old access control policies.

Third, Antidote provides a highly available form of transactions, where reads observe a causally-consistent snapshot of the database and writes are made visible atomically. Unlike standard transactions, write-write conflicts are solved by merging the concurrent update. Applications can leverage these highly-available transactions to guarantee that a set of updates is made visible atomically.

Fourth, Antidote provides support for efficiently enforcing numeric invariants, such as guaranteeing that the value of a counter remains larger than 0. To this end, it includes an implementation of a Bounded Counter CRDT [20], a shared integer that must remain within some bounds. The implementation uses escrow techniques [21] for allowing an operation to execute in a replica without coordination in most cases.

Finally, associated with Antidote, we have developed a set of tools to verify whether an application can execute correctly under weak consistency, and when this

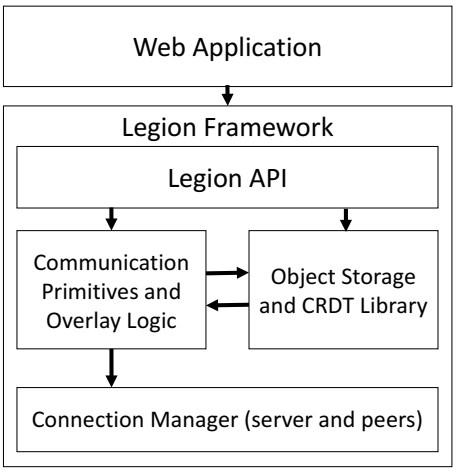

Figure 1.1 The Legion architecture (adapted from [24])

is not the case, what coordination is necessary. These tools are backed by a principled approach to reason about the consistency of distributed systems [22].

Antidote is designed to be deployed in a set of geo-distributed data centers. Within each cluster, data is sharded among the servers. Data is geo-replicated across data centers. The execution of transactions in Antidote, and the replication of updates across data centers, is controlled by Cure [23], a highly scalable protocol that enforces transactional causal+ consistency (combining CRDTs for eventual consistency, causal consistency and highly available transactions).

Legion. Legion [24] is a new framework for developing collaborative web applications that transparently leverage on the principles of edge computing by enabling direct browser-to-browser communication. Legion was implemented in *javascript* and it uses the *Web Real-Time Communications* (https://webrtc.org) to establish direct communication channels among web application users. At its core, Legion enables applications to transparently replicate, in the form of CRDTs, relevant application state in clients. Clients can then modify the application state locally, and through the use of hybrid gossip mechanisms, synchronize directly among them, without the need to go through the web application server. The server however is still used both to ensure the durability of the application state, but also to assist in the operation of Legion, namely to simplify the task of creating the initial webRTC connections among clients when they enter the application.

A simplified architecture of Legion is illustrated in Figure 1.1. Legion can be used by a web application simply by importing a javascript script. This script provides the application access to the *Legion API*. The API exposes to the application the ability to manipulate data objects that can be used to model the application state. These data objects include records, counters, lists, and maps. All of these objects are internally represented by Legion through CRDTs which simplifies the the direct synchronization among clients of shared application state. This is provided by an extensible CRDT Library that is part of the *Object Store* component of Legion. The synchronization of objects among clients (and that of a subset of clients with the server to ensure durability) is transparently managed by the Object Store.

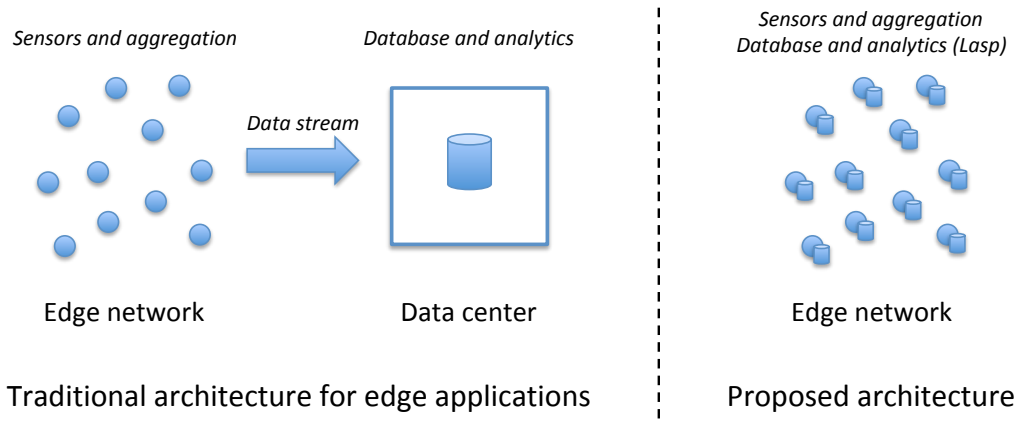

Figure 1.2 Proposed architecture for edge applications using Lasp

To guide the synchronization process, Legion leverages on an unstructured overlay network, whose construction is guided by the principles of hybrid gossip, and takes into consideration the relative distance of each client among them. This allows clients to mostly interact and synchronize with clients that are in their vicinity. While the typical use case in Legion is to have clients interacting through the manipulation of shared data objects, web applications also have access to communication primitives that enable them to disseminate messages among the currently active clients of the application in a decentralized fashion. This is achieved by a gossip-based broadcast protocol that operates on top of the legion overlay network.

Finally, Legion also takes into account security, by ensuring that before clients can start to replicate and manipulate application data objects they authenticate on a server. Moreover, Legion exposes an adapter API, that allows developers to integrate their Legion-backed applications with existing backends. The framework provides adapters to the Google Real Time API[6]. These adapters allow the developers to leverage this backed to do any combination of the following: authentication and access control, data storage for durability, and support to the WebRTC signaling protocol required to create webRTC connections among browsers. More details on the design and operation of Legion can be found in [24]. Legion is open source and available, along side some demo applications through https://legion.di.fct.unl.pt.

Lasp. The Lasp language and programming system [25] was designed for application development on unreliable distributed systems, and in particular for edge computing. Lasp allows developers to write applications by composing CRDTs, as explained above [18]. In addition to composition, Lasp also provides a monotonic conditional operation that allows executing application logic based on monotonic conditions on CRDTs. The Lasp implementation combines a programming layer based on synchronization-free computing with a communication layer based on hybrid gossip. This makes the implementation highly resilient and well-adapted to edge networks.

Many of today's edge applications use the cloud as a database to store data coming from the edge. By using Lasp as their database, such applications can be trans-

[6]https://developers.google.com/google-apps/realtime/application

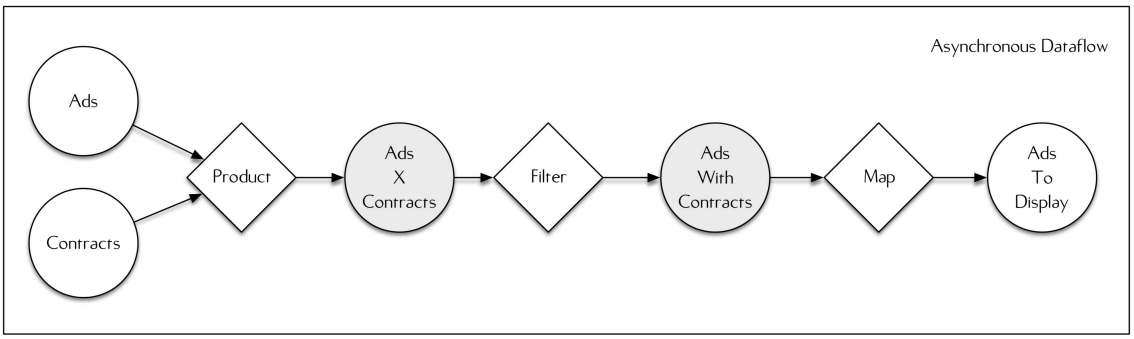

Figure 1.3 A Lasp computation to derive the set of displayable advertisements in the advertisement counter scenario. On the left, Ads and Contracts give information for the advertisements, including how many times they have been displayed, and their contracts, including the threshold for each advertisement. On the right are the advertisements that can be displayed. All data structures are sets, similar to database relations, and the computation is similar to an incremental SQL query.

lated to fully run on the edge (see Figure 1.2). This cannot be done with traditional cloud databases since they are not designed to run on unreliable edge networks. In the proposed architecture, the edge network runs everything: the sensors and aggregation software on individual edge nodes, and the database (Lasp) on all edge nodes. Analytics computations can be run either as an internal Lasp computation or external to Lasp on individual nodes, using Lasp just as a database.

Example Lasp program. A typical application for Lasp is the scenario of advertisements counter that counts the total number of times each advertisement is displayed on all client mobile phones, up to a preset threshold for each. Figure 1.3 defines graphically part of the Lasp program for this application. The actual code is a straightforward translation of this graph. The application has the following properties:

- Replicated data: Data is fully replicated to every client in the system. This replicated data is under high contention by each client.
- High scalability: Clients are individual mobile phone instances of the application, thus the application should scale to millions of clients.
- High availability: Clients need to continue operation when disconnected as mobile phones frequently have periods of signal loss (offline operation).

This application can be implemented completely on the edge, as explained previously, or partly on the cloud. For this application we have demonstrated the scala-

bility of the Lasp prototype implementation up to 1024 nodes by using the *Amazon* cloud computing environment to simulate the edge network [26].

1.1.5 Future Directions

Building additional tools and support for a new generation of ultrascale edge applications is quite relevant and challenging. The varied nature of edge computing environments, which can combine small private clouds and data centers, specialized routing equipment and 5G towers, users desktops, laptops and even cellphones, to small things sensors and actuators, makes it a daunting task to build a single runtime support that can efficiently operate on all such devices and deal with their heterogeneity.

While we presented a set of tools and frameworks that can ease the development of ultrascale edge computing applications and services, these do not cover all possible execution scenarios. That path to build such support requires not only the development of specialized runtimes for different edge settings, but also devising standard protocols and data representation models that allow the natural integration of different runtimes in a cohesive and effective edge architecture.

Current solutions for data replication and management are also unsuitable for the ultrascale that one is expected to find in emerging edge computing applications. The use of CRDTs to address the requirements of data management in this setting presents a viable approach. However, further efforts have to be dedicated in designing new and efficient synchronization mechanisms that can naturally adapt to the heterogeneity of the execution environment.

References

[1] Armbrust M, Fox A, Griffith R, et al. A view of cloud computing. Communications of the ACM. 2010;53(4):50–58.

[2] Leitão J, Pereira J, Rodrigues L. Epidemic Broadcast Trees; 2007. p. 301–310.

[3] Leitão J, Pereira J, Rodrigues L. HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. In: Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on; 2007. p. 419–429.

[4] Shapiro M, Preguiça N, Baquero C, et al. Conflict-free replicated data types. INRIA; 2011. RR-7687.

[5] Almeida PS, Shoker A, Baquero C. Delta state replicated data types. Journal of Parallel and Distributed Computing. 2018;111:162–173.

[6] Carlos Baquero PSA, Shoker A. Making Operation-Based CRDTs Operation-Based. In: Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings; 2014.

p. 126–140. Available from: http://dx.doi.org/10.1007/978-3-662-43352-2_11.

[7] Bonomi F, Milito R, Zhu J, et al. Fog Computing and Its Role in the Internet of Things. Proceedings of the first edition of the MCC workshop on Mobile cloud computing. 2012;p. 13–16. Available from: http://doi.acm.org/10.1145/2342509.2342513$\backslash$npapers2://publication/doi/10.1145/2342509.2342513.

[8] Yi S, Li C, Li Q. A Survey of Fog Computing: Concepts, Applications and Issues. In: Proceedings of the 2015 Workshop on Mobile Big Data. Mobidata '15. New York, NY, USA: ACM; 2015. p. 37–42. Available from: http://doi.acm.org/10.1145/2757384.2757397.

[9] Verbelen T, Simoens P, De Turck F, et al. Cloudlets: Bringing the cloud to the mobile user. In: Proceedings of the third ACM workshop on Mobile cloud computing and services. ACM; 2012. p. 29–36.

[10] Fernando N, Loke SW, Rahayu W. Mobile cloud computing: A survey. Future generation computer systems. 2013;29(1):84–106.

[11] Hu YC, Patel M, Sabella D, et al. Mobile edge computingA key technology towards 5G. ETSI white paper. 2015;11(11):1–16.

[12] Cisco. Cisco IOx Data Sheet; 2016. Available from: http://www.cisco.com/c/en/us/products/collateral/cloud-systems-management/iox/datasheet-c78-736767.html.

[13] Dell. Dell Edge Gateway 5000; 2016. Available from: http://www.dell.com/us/business/p/dell-edge-gateway-5000/pd?oc=xctoi5000us.

[14] Milojicic DS, Kalogeraki V, Lukose R, et al. Peer-to-peer computing. 2002;.

[15] Jelasity M, Montresor A, Babaoglu O. Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems (TOCS). 2005;23(3):219–252.

[16] Akyildiz IF, Su W, Sankarasubramaniam Y, et al. Wireless sensor networks: a survey. Computer networks. 2002;38(4):393–422.

[17] Gilbert S, Lynch N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. SIGACT News. 2002 Jun;33(2):51–59. Available from: http://doi.acm.org/10.1145/564585.564601.

[18] Meiklejohn C, Van Roy P. Lasp: A Language for Distributed, Coordination-Free Programming. In: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP 2015); 2015. p. 184–195.

[19] Carvalho N, Pereira J, Oliveira R, et al. Emergent Structure in Unstructured Epidemic Multicast. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07). Edinburgh, Scotland, UK; 2007. p. 481 – 490.

[20] Balegas V, Serra D, Duarte S, et al. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. Montréal, Canada; 2015. p. 31–36. Available from: http://lip6.fr/Marc.Shapiro/papers/numeric-invariants-SRDS-2015.pdf.

[21] O'Neil PE. The Escrow Transactional Method. ACM Trans Database Syst. 1986 Dec;11(4):405–430. Available from: http://doi.acm.org/10.1145/7239.7265.

[22] Gotsman A, Yang H, Ferreira C, et al. 'Cause I'M Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 2016. New York, NY, USA: ACM; 2016. p. 371–384. Available from: http://doi.acm.org/10.1145/2837614.2837625.

[23] Akkoorath DD, Tomsic A, Bravo M, et al. Cure: Strong Semantics Meets High Availability and Low Latency. INRIA; 2016. RR-8858.

[24] van der Linde A, Fouto P, Leitão J, et al. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In: Proceedings of the 26th International Conference on World Wide Web. WWW '17. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee; 2017. p. 283–292. Available from: https://doi.org/10.1145/3038912.3052673.

[25] Lasp: The Missing part of Erlang distribution;. Accessed: 2018-04-27. http://www.lasp-lang.org.

[26] Meiklejohn C, Enes V, Yoo J, et al. Practical Evaluation of the Lasp Programming Model at Large Scale. In: Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP 2017); 2017. p. 109–114.

B References

[1] Carlos Baquero and Francisco Moura. Specification of convergent abstract data types for autonomous mobile computing. Technical report, Departamento de Informática, Universidade do Minho, 1997.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pages 229–241, Monterey CA, USA, November 1994. ACM.

[4] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 394–401, Hong Kong, May 1996.

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

[6] Sreeja S Nair. Evaluation of the CEC (Correct Eventual Consistency) Tool. Research Report RR-9111, Inria Paris ; LIP6 UMR 7606, UPMC Sorbonne Universités, France, November 2017.

[7] Marc Shapiro, Annette Bieniusa, Peter Zeller, and Gustavo Petri. Ensuring referential integrity under causal consistency. In Sebastian Burckhardt and Marko Vukolić, editors, *W. on Principles and Practice of Consistency for Distr. Data (PaPoC)*, page 5, Porto, Portugal, April 2018. Euro. Conf. on Comp. Sys. (EuroSys), Assoc. for Computing Machinery.

[8] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique (Inria), Rocquencourt, France, November 1992.

[9] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Dist. Sys. (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, October 2011. Springer-Verlag.

[10] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 233–242, Vancouver, BC, Canada, August 1984.