LIGHTKONE

Lightweight computation for networks at the edge

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

D3.1: Initial Runtime Edge Computing System

Deliverable no.:	D3.1
Title:	Initial Runtime Edge Computing System
Due date of deliverable:	January 15, 2019
Actual submission date:	January 15, 2019
Lead contributor:	INESC TEC
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
15/01/2019	2.0	Revised version ready for submission	INESC TEC
15/10/2018	1.1	Starting revision	INESC TEC
09/02/2018	1.0	Final Version	INESC TEC
27/01/2018	0.3	Incorporating internal reviews	UPC & NOVA
20/12/2017	0.2	Content structure	INESC TEC
15/12/2017	0.1	1st draft with outline and ToC	INESC TEC

Contributors:

Contributor	Institution
Ali Shoker	INESC TEC
João Marco Silva	INESC TEC
Georges Younes	INESC TEC
Manuel Bravo	UCL
Christopher Meiklejohn	UCL & IST/INESC-ID
Annette Bieniusa	TUKL
Deepthi Akkoorath	TUKL
Igor Zavalyshyn	IST/INESC-ID
Paulo Sérgio Almeida	INESC TEC
Vitor Enes	INESC TEC
Carlos Baquero	INESC TEC
Sébastien Merle	STRITZINGER
João Leitão	NOVA
Pedro Ákos Costa	NOVA
Carla Ferreira	NOVA
Nuno Preguiça	NOVA
Gonçalo Cabrita	NOVA
Roger Pueyo Centelles	UPC
Marc Shaprio	SU
Giorgos Kostopoulos	GLUK
Brad King	Scality
Dimitrios Vasilis	SCALITY
Peer Stritzinger	STRITZINGER
Roger Pueyo Centelles	UPC
Felix Freitag	UPC

Contents

Chapter 1

Executive Summary

This deliverable (D3.1) presents the data and communication abstractions and components that can be used to build a generic edge computing runtime, as well as the cross-cutting concerns of LightKone work packages (WPs) including the LightKone Reference Architecture: LiRA. LightKone advances state of the art of edge/fog computing by providing highly available and scalable replicated data management strategies. Core to these solutions are recent advances in Conflict-free Replicated DataTypes (CRDTs) and as well in their supporting dissemination middleware layers. Consequently, in this deliverable, corresponding to work package 3 (WP3), we convey contributions on CRDTs as well as on the underlying dissemination layer. The report also presents a preliminary security analysis to LightKone use-cases. The core contributions can be summarized as follows:

LightKone Reference Architecture (LiRA). LiRA represents the reference architecture that governs the LightKone contributions across work packages. LiRA is shown to be compliant with well known state of the art edge/fog computing reference architectures (as Open Fog), but complements them through addressing the data management layer and application semantics. LiRA is composed of several artifacts that span a wide spectrum of edge/fog networks and scenarios including, what we also define, *heavy*, *light*, and *hybrid* edge. LiRA satisfies the needs of LightKone's use-cases as well as other ones as we show in this report.

Data and communication abstractions. We extended the existing CRDT datatypes, both state-based and operation-based, both by providing more comprehensive catalogues of datatypes, and through improving their efficiency on the data and communication levels (via reducing the meta-data stored and disseminated at the edge). We have introduced hybrid gossip-based group membership libraries supporting diverse and dynamic edge network topologies as well as improving the Distributed Erlang OLP library. Finally, we introduced new scalability solutions based on partial replication for heavy edge networks, and other scalable counter datatypes avoiding identity explosion in large edge networks. The corresponding developed software are also presented.

Security analysis. Finally, we provide a preliminary security analysis to LightKone use-cases.

Chapter 2

Introduction

With the immense volumes of data generated and computed at the cloud data centers, there is a need to move part of storage and computation towards the edge of the network. This brings many benefits like reducing the volumes of data traveling to cloud data centers and avoiding potential bottlenecks, improving the availability of services, privacy, etc. This comes at the price of new data and communication challenges at the entire edge software stack, among them, scalability, heterogeneity, resilience, and security. In this work package, i.e. WP3, we study and provide the building blocks for a generic edge computing runtime with special emphasis on data abstractions and communication protocols, while including the cross-cutting concerns of other work packages. In particular, this deliverable D3.1 presents the LightKone Reference Architecture (LiRA) that governs LightKone's artifacts and components. The deliverable then presents the basic data abstractions and their supporting communication protocols through filling some of the existing gaps in both functional and nonfunctional properties supporting edge applications. We demonstrate how these components contribute to building LiRA's artifacts. Finally, we present the use-case security requirements to address in the next deliverable.

2.1 Motivations, approach, and methodology

Motivations. There is a notable interest in the Edge/Fog computing paradigm which lead to several academic and industrial projects targeting the entire software edge computing stack. However, as we show in Chapter 7, state of the art (SOTA) solutions revolve around the following pattern: edge devices stand as intermediary relays that summarize the collected data from the source, pushing them towards the data center (or other edge layers), and/or feeding the processed data back to the edge of the network (e.g., actuators). Although interesting, this does not cover the scenarios where edge devices can share and process data (through replication) when the cloud data center is unreachable or the cost of reaching it is high. The default option could be to use a replication scheme with classical strong coordination, e.g. Paxos or Raft, however, these solutions are likely not suitable given the dynamics and connection brittleness often present in edge networks. Conditions that may lead to the violation of the semantics and properties of edge applications, namely their expected availability.

Approach. In LightKone, and WP3 in particular, we tackle this challenge through following a relaxed data consistency model where data and computation are replicated over loosely coupled edge machines on the premise that replicas will eventually converge. Consequently, we build on the success of Conflict-free Replicated DataTypes (CRDTs) [84] which were developed in EU FP7 Syncfree project [34], and extend them to support edge computing. CRDTs are data abstractions that are mathematically guaranteed to converge when replicas eventually apply the same (likely concurrent and commutative) operations. This also requires a supporting communication layer that can eventually deliver operations or changes across replicas respecting specific requirements like causality. We address this layer through developing convenient causal broadcast middlewares and anti-entropy gossip-based protocols.

Methodology. In his work package we try to provide the distributed data and communication abstractions necessary to build generic edge computing runtimes that are highly available, convergent, and reliable. Since the target is generic edge computing, we follow two complementary directions. The first is to directly address the requirements of the LightKone use cases that eventually manifest in the artifacts shown in LiRA after further development specific to other work packages, namely WP5 (Light Edge) and WP6 (Heavy Edge). However, since LightKone use cases are only a sample of the edge application space, we believe it is necessary to provide edge components with features that support generic edge runtimes beyond LightKone's use cases—as long as they are feasible for other edge applications. We believe this is consistent with the nature of Research and Innovation projects in which a wider applicability is desired.

2.2 Contributions

We summarize the key contributions of WP3 for year 1 (Y1). The contributions follow a plan that roughly follows the LightKone proposal, and presented in Chapter 4. These contribution are summarized as follows.

2.2.1 LightKone Reference Architecture (LiRA)

We present the LightKone Reference Architecture (LiRA) that is designed for the support of applications, computations and data in fog and edge networks. LiRA is compatible with state of the art (SOTA) reference architectures but complementary as it focuses in application-level data management, mainly CRDTs [84] and communication support. It is thus possible in LiRA to share data at the same edge level as well as across fog levels. LiRA presents in the Architecture View the several artifacts developed in LightKone demonstrating their coverage of a wide range of fog networks and hardware. These artifacts are built using software components and libraries developed in cooperation with WP3, and presented in the Component View. These components also serve as building blocks for generic edge/fog runtimes beyond LightKone. On the other hand, we demonstrate in the Use Case View the feasibility of LiRA to various LightKone use cases through explaining the workflow and the interplay between its artifacts in each use case. Finally we present the relation to SOTA edge/fog architectures, and we present the terminology and classification we adopt in LiRA.

2.2.2 Data Abstractions at the Edge

Although proven successful in geo-replication, the current state of the art (SOTA) of CRDTs [84, 92] cannot satisfy the needs of the edge for several reasons attributed to the hostile edge networks and constrained devices, e.g., efficiency, scalability, heteroginity, etc. In this report, we focus on efficiency, and we aim at further progress in future reports. In particular, in Section 4.2, we present optimizations for several variants of state-based and op-based CRDT models through reducing the meta-data shipped over the network and stored in devices. Some of these contributions involved hands-on optimizations for CRDT implementations embedded in LightKone artifacts, as in AntidoteDB [35] (presented in detail in WP6). In addition, we tried to fill the gap of missing datatype specifications by providing a portfolio for various variants of counters, registers, sets, maps; and support important and challenging operations as "reset" [7, 17].

2.2.3 Communication support for data at the Edge

The data management techniques and datatypes mentioned above assume the presence of an underlying dissemination layer with properties that support general cloud and edge applications (e.g., causality), as well as networks (e.g, scalability and dynamicity). In particular, the work on advanced Pure op-based CRDTs [17] assume the presence of a causal middleware that is efficient and supports "causal stability" which is novel to SOTA causal middlewares [17, 18, 88]. In Section 4.3, we present a middleware that supports these features. On the other hand, all the dissemination protocols used in the previous sections are implemented using Distributed Erlang that has known limitations in parallelism and no support for cluster topologies (by assuming *full mesh*). We developed a communication library called Partisan in a previous project (FP7 Syncfree [34]) that implements two group membership protocols, i.e., Plumtree [86] and HyParView [63], that are efficient hybrid gossip protocols. In LightKone, we developed Partisan further to support edge networks and edge applications. In particular, we now support: dynamic network topologies, different application patterns, sending in multiple channel. At a lower protocol layer, we are working with Ericsson to develop the Distributed Erlang library to support generic networks with routing instead of relying on full connected mesh, avoiding the scalability limitations of the full mesh (where the number of connections in the cluster grows quadratic to the number of nodes).

2.2.4 Scalable data management at the edge

Data scalability is an essential feature to support edge networks and applications. There are at least two scalability dimensions of particular interest at the edge: storage and network size. The former differs from classical cloud systems given the relatively limited storage capacities of edge devices, even those at the heavy edge like micro data centers [55, 68, 95]. A natural technique to address this challenges is to use partial replication (a.k.a., partitioning or sharding). SOTA data partitioning is inadequate to edge systems due to the overhead of metadata and voluminous payloads disseminated especially when some properties, like causal consistency, are needed [66, 67, 111]. In Section 4.4, we introduce two solutions to address data partitioning through reducing the meta-data disseminated, as explained next in Saturn, or even avoid sending the payload if

unnecessary, as explained next in nonuniform partial replication. The other dimension is addressing the increasing network size likely in edge networks and applications. In particular, highly available datatypes as CRDTs can only scale to few tens of nodes due to the incurred metadata overhead [7, 17, 84]. To that end, we introduce two techniques that provide highly scalable counters using hierarchical containment (i.e., Handoff Counters) or transient identity borrowing (i.e., Borrow Counters).

2.2.5 Software deliverables

In Chapter 5, we present the software deliverables, libraries, and components in which the contributions presented in this report appear. Most of these software are direct artifacts that show in LiRA or used as backend components and libraries. Since this work package is meant to provide the support to build generic edge computing runtimes, we believe that developing fine-grained components is crucial to increase the impact of LightKone's work on external edge computing platforms. Indeed, although LiRA perfectly fits the set of LightKone use-cases, the latter represents a sample edge computing set of applications, and thus addressing more use-cases may require building other edge computing runtimes. To this end, the components provided in this deliverable can be used in building new edge runtimes or integrated in existing ones to leverage the technology LightKone provides. All presented software are made available on Github.

2.2.6 Security analysis of use-cases

Although important, LightKone's contribution on security is limited to analyzing the security threats and requirements of LightKone use-cases (presented in deliverable D2.1). The corresponding solutions can then be selected by using standard off-the-shelf security methods and tools or through developing novel solutions. Since LightkKone's focus is mainly on data managements and communication, Chapter 6 only highlights the open issues and potential threats, recommend off-the-shelf security measures, and develop new solutions to only part of these threats—as defending against DoS attacks in WP5. The analysis concluded that the majority of use-cases (UCs) share requirements that involve balancing data integrity, confidentiality, availability and authentication with some kind of constraint in such entities. Another conclusion is that most of the threats are often on the light edge part. The analysis also gave example of some research topics might benefit all use-cases: lightweight cryptography for Light-edge environments, Homomorphic encryption algorithms, cross-platform software sandboxing, and DoS identification and prevention.

2.2.7 Exploratory research

Finally, we present the exploratory research work that is not at the core of LightKone, though related. These works have the potential of more exploration or inspiration in the future. For instance, we present the Single-Writer principle that discusses the cases where some CRDTs may not incur concurrency. We also include two security works for Privacy-aware IoT Data Management at the light edge and Byzantine resilient protocol for heavy and possibly light edge.

2.3 Relation to other WPs

Work packages addresses the cross-cutting concerns across all other work packages and thus it depicts a natural overlap with most of them. First, LiRA describes the interplay of the various artifacts developed in work packages WP5, WP6, and with the help of WP4. Second, the work developed in WP3 is focused at generic components and techniques to build edge runtimes; the artifacts developed in WP5 and WP6 demonstrate practical examples how to use these components to build light and heavy edge runtimes. For instance, the designed CRDT libraries in this deliverable represent the building blocks of the data models in D4.1 and implementations in D5.1 and D6.1. In particular, AntidoteDB is developed in WP6 and uses the optimized op-based CRDTs, whereas Lasp and Legion developed in WP4 and WP5 uses the delta- and state-based CRDTs. Similarly, some of the communication protocols developed in WP3 like Partisan is being used as the underlying communication layer of Lasp. Third, non-uniform replication is being used in AntidoteDB and explored in Yggdrasil in WP6 and WP5, respectively. Fourth, the work on improving the Erlang VM overlaps with WP4, 5, and 6 since all artifacts are built in Erlang as a lower layer. Finally, the work on security analysis and application support in this deliverable builds on the use-cases presented in D2.1, and discusses the security measures that should respect the requirements in WP1.

2.4 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:

- Introduced the Lightkone Reference Architecture (LiRA).
- Introduced a literature review for state of the art (SOTA) works showing how the WP advances beyond it.
- Introduced the plan we followed in Year 1 and the future plan to achieve the mentioned milestones.
- Revised all sections to improve presentation with referencing to the SOTA, framing the contribution, how it advances beyond SOTA according to the plan, and the respective future plan.
- Removed all documents in the appendix and replaced them with annotated bibliography for the convenience of the reader; and dedicated a section for dissemination activities carried in the context of this work package.

2.5 Organization of the Report

The rest of the report is organized as follows:

Chapter 2: LightKone Reference Architecture (LiRA) presents the LightKone Reference Architecture and the terminology followed in LightKone.

Chapter 3: Plan and Progress presents the plan we followed in WP3 and the work progress considering the data and communication abstracts.

Chapter 4: Software Deliverables conveys the software deliverables developed in WP3 together with other WPs.

Chapter 5: Security Analysis of Use-cases presents the preliminary LightKone use-case security analysis.

Chapter 6: State of The Art presents the state of the art of the areas covered in this report.

Chapter 7. Exploratory Research presents other research works that are minor to LightKone.

Chapter 8: Annotated Publications & Dissemination presents a synopsis of the publications and dissemination in WP3.

Chapter 3

LightKone Reference Architecture (LiRA)

3.1 Introduction

The LightKone project aims to develop solutions for supporting the creation of systems and applications that can execute general-purpose computations in edge networks with the goal of providing high availability, responsiveness, and fault tolerance. The LightKone project takes a broad view of edge, close to a unified ”fog” vision encompassing the whole spectrum of devices and networks from core cloud computing (based on heavyweight data centres with abundant storage, computation, network and support resources), to points-of-presence, and all the way to lightweight user devices (featuring mobility, on-off presence, scarce resources, and absence of expert administration).

The LightKone Reference Architecture (LiRA) provides an overview of the architecture and technology that can be used for developing systems and applications that run in edge networks. LiRA consists of a set of general-purpose components, with a uniform semantics, that together support fog and edge computing functionality, combining high availability and correctness. This includes highly-available distributed data types (e.g., CRDTs), resilient and durable distributed storage (e.g., key-value stores), highly-available and consistent data-sharing models (e.g., Transactional Causal Consistency), security, support for both OLAP and OLTP-style computation placed at strategic locations in the network, discovery, etc. They are designed to function correctly and to scale without friction, in the presence of slow, unreliable and dynamic networks, which are inherent features of the edge.

We structured the presentation of LiRA according to the following “Views”:

- **Architecture View:** this view presents the artefacts developed in the context of LightKone, covering the edge computing spectrum, and how they can be combined for developing edge computing solutions.
- **Component View:** this view presents the generic components that can be used in edge computing solutions, and that are used, in particular, in the artefacts developed in the context of LightKone.
- **Use-case View:** this view depicts how the artefacts and components interplay to serve the purposes of each use-case.

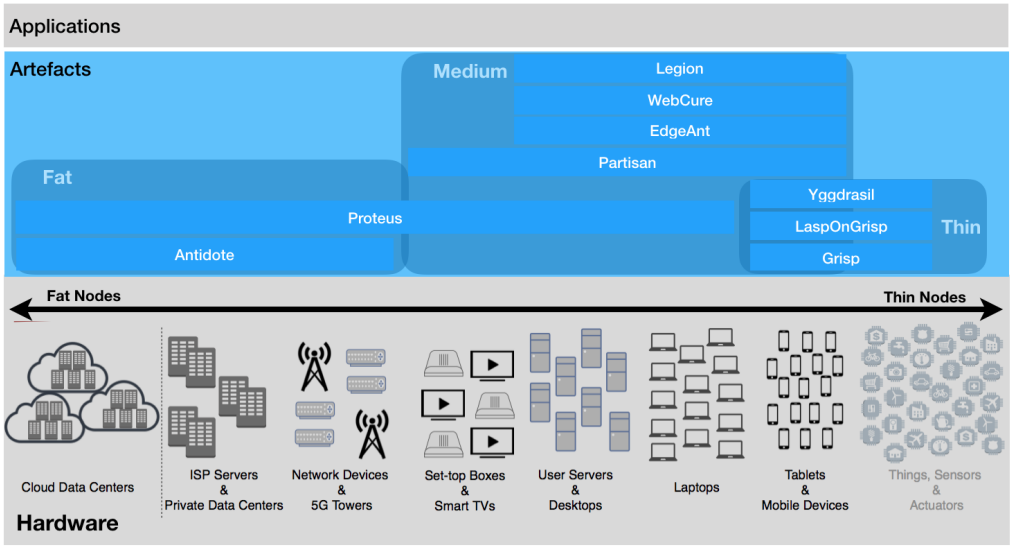


Figure 3.2.1: Architecture View.

For the rest of this chapter, we present these Views, and then we present the edge/fog system models and terminology followed in LightKone. We finally discuss the relation of LiRA to state of the art fog/edge architectures.

3.2 Architecture View

The Architecture Viewpoint (Figure 3.2.1) presents the artefacts developed in the context of the project and how they address the challenges posed by developing applications for edge computing scenarios. We broadly classify the artefacts in Heavy, Light, and Thin according to the settings where they were designed to run, with each setting posing specific challenges that need to be addressed using different techniques. We now give a brief overview of the type of artifacts developed and how they relate (the list of artefacts is not exhaustive), and a summary of these artefacts is presented in Table 3.2.1 (at the end of this chapter).

The Heavy-Edge artefacts were designed to run in nodes with substantial storage and computation capacity (like nodes in public and private cloud infrastructures and ISPs). In this context, we focused in two types of artefacts: replicated data management systems and indexing services. A data management system will be used by applications to store application data. The key requirements of such system is to provide high availability, fault-tolerance and low latency to clients. AntidoteDB is a highly available geo-distributed database that provides address these requirements. The indexing services can be used by applications to provide efficient search of the application information, which can be potentially stored in different databases. Proteus is a system that can be used for such purpose. When necessary, we expect that applications will rely in other types of services, such as messaging services like Apache Kafka, or data processing frameworks like Apache Spark or Apache Storm to fulfill their needs.

The Light-Edge artefacts were designed to run in personal nodes with different storage and processing capacities, including smartphones, laptops and users' servers. In this context, we focused mainly on artefacts that support data sharing and communication

among devices. The key challenges in this context are to provide low latency of interaction among devices, high availability despite node disconnection (including disconnected operations) and address the specific needs of different applications. EdgeAnt provides a cache that allows applications to access data stored in Antidote with low latency, by relying on the data stored in the local cache. It additionally provides support for disconnected operation. WebCure has similar goals, but it is designed to support web applications what run in browsers, thus having to address the challenges posed by running in the browsers' constrained environment. Legion extends WebCure goals by supporting direct interaction among clients, for low latency in interactions among clients. It also supports interaction with different cloud services. For applications that cannot run on top of the data sharing services provided by the LightKone artefacts, we have developed Partisan, a communication middleware that can be used for exchanging data among multiple nodes. Partisan can be used by applications with different requirements, providing an efficient communication substrate that simplifies the development of such applications.

The Thin artefacts were designed to run in the small devices with low memory and storage capacity, including sensor nodes, "things" and mobile devices. In this context we focused in the following types of artefacts. First, communication services for propagating information among nodes of the systems. Yggdrasil provides a generic framework for designing distributed protocols for ad-hoc networking. For example, we have used Yggdrasil for designing an aggregation protocol, that can be used for collecting information from sensors and eventually propagate it to an external service (e.g. Antidote or one of the Light-Edge artefacts). Second, software for embeddable devices. Grisp software stack provides efficient communication for application running in ErlangVM in the Grisp nodes. Finally, data sharing services embeddable devices. In this context, LaspOnGrisp provides a key-value store that allows applications running in embeddable devices an high-level abstraction for data sharing.

3.3 Component View

The Component viewpoint presents the essential building blocks that are helpful to support general-purpose edge computing. These building blocks can be combined and used in the implementation of a specific artifact (or sub-system), for providing a general-purpose service, such as geo-distributed communication, notification, storage, searching, computation, etc.

We classify the building blocks that we have been developing in three main groups: Communication, Data Management, and Computation. Tables 3.3.1, 3.3.2, and 3.3.3 list some of the key components implemented in the context of the LightKone project, with a brief description, previous state of the art, novelty, implementation artefact, and reference to the section for more reading.

Table 3.3.1: Communication components.

Component	Description	Previous SOTA	Contribution	Artefact	Reference
Causal Delivery and stability middleware	middleware for causal consistent systems and op-based CRDTs	Redundant meta-data in causal delivery	Reduced meta-data in causal delivery; causal stability concept	None	D3.1
Causal Delivery and stability middleware	middleware for causal consistent systems and op-based CRDTs	reduced meta-data in causal delivery; causal stability concept;causality issues in callback-based and independent threads/processes.	End-to-End Causal Delivery; Correct tagging; Implementing Causal Delivery and Stability	None	D3.2
Distributed Communication	Edge-tailored alternatives for distribution layer for Erlang.	Plumtree, HyParView; Erlang distribution	Partisan: Hybrid gossip-based with different net topologies and various clusters; mesh-based EVM.	Lasp, LaspOnGrisp	D3.1
Distributed Communication	Edge-tailored alternatives distribution layer for Erlang.	Partisan: Hybrid gossip-based with different net topologies and various clusters; mesh-based EVM.	Partisan channel-based full mesh back-end; evaluation (# of nodes scalability); EVM API with custom implementations	Lasp, LaspOnGrisp	D3.2

Table 3.3.2: Data management components.

Component	Description	Previous SOTA	Contribution	Artefact	Reference
State CRDT	State-based data management for relaxed consistency at the edge	Not generic enough; few datatypes	Generic framework; many datatypes; join-decomposition	Legion, Lasp	D3.1
Op CRDT	Operation-based data management for relaxed consistency at the edge	Non-edge-tailored datatypes; few datatypes; not generic	Generic framework; optimized edge-tailored datatypes; support resets; many datatypes; resettable counters	AntidoteDB	D3.1
Op CRDT	Operation-based data management for relaxed consistency at the edge	Generic framework; optimized datatypes; support resets; many datatypes	Compression at the source	None.	D3.2
Scalable Counters (Handoff and Borrow)	Datatypes scalable with the number of edge nodes or dynamicity	blocking sync datatypes; ID explosion	scalable counters using hierarchical trees; transient IDs for counters; single writer principle	None	D3.1
Saturn	Partial replication (sharding) meta-data handling with causality support	Causal multicast protocols; Cure protocol for causal delivery	Reduced metadata propagation for enforcing causality; timely delivery of updates	None	D3.1
C3	Improved Partial replication (sharding) meta-data handling with causality support	Saturn	Improved concurrency on the server	None	D3.2

Table 3.3.3: Computation components.

Component	Description	Previous SOTA	Contribution	Artefact	Reference
Mirage	Protocol for aggregation in ad-hoc networks	Distributed aggregation protocols	Efficient handling of variation of input values	Mirage @ Yggdrasil	D5.2
Computation CRDTs	CRDTs for which the state is the result of a computation over the executed operations (e.g. aggregation results), adopting the non-uniform replication model	Distributed aggregation protocols	Non-uniform replication model; integrates computations with the storage	Antidote-DB	D3.1
Distributed analytics middleware	Middleware for analytics computations on federated data stores	Apache Spark, Multi-store systems	Bidirectional data-flow computations using materialized views. Modular distributed architecture that enables flexible data and computation placement.	Proteus	D6.2

3.4 Use-case View

The use-case view shows how the artefacts and components developed in the context of the project, and presented in the other views, can be combined to serve the purposes of each use-case. Being diverse, each use-case is depicted in a separate Use-case View highlighting the useful artefacts that could be used to address its requirements and technological needs.

3.4.1 Distributed monitoring for community network (Guifi.net)

Guifi.net has built IP communication network where a large part of the network is formed by many interconnected commodity wireless routers and some parts are build with fibre optics connectivity.

For the management of the network, these commodity wireless routers need to be monitored in order to confirm correct operation, monitor traffic, and detect any issues with the connection and operation of these routers.

For the monitoring of these routers, hundreds of servers are run inside of Guifi.net by individuals and diverse organizations. These monitoring servers, often also used for other tasks, connect periodically to these routers and applying SNMP (Simple Network Management Protocol), obtain operational information of the routers. The operational information collected by the monitoring servers can be visually inspected in the Guifi.net Web.

As detailed in D2.1, there are several important limitations in the current implementation of the monitoring system use case of Guifi.net. Among these limitations is the fact of having single point of failures in several functions. For instance, a router is only monitored by a single server. The assignment is done once manually and not updated. The monitoring data is stored only locally in each monitoring server for graphical visualization without further processing and analysis. As a consequence, once a monitor is disconnected, the information about a router is lost. Disconnection can be temporarily, e.g. network or server issues, or permanently.

The new implementation of the distributed community networking use case aims to improve the current situation with several features: First, we aim to monitor each router

by a set of monitoring servers. Failures in one of the monitoring servers will still enable to monitor the router by the other servers. In addition, the assignment of monitoring servers to routers will be made dynamic, i.e., in the regime of permanent operation, the assignment can be updated by the servers in a decentralized way as to contextual information (e.g. temporary network situation, server load, relevance of the router). Secondly, we aim to store and merge the monitored information. Each router is monitored by several servers. The data collected by each monitoring server is to be merged and stored to enable further processing and analysis. The new implementation brings the benefits of a more robust system and opens the door to conduct a deeper network analysis, which in turn is important for the mechanism which Guifi uses for the sustainability of the network.

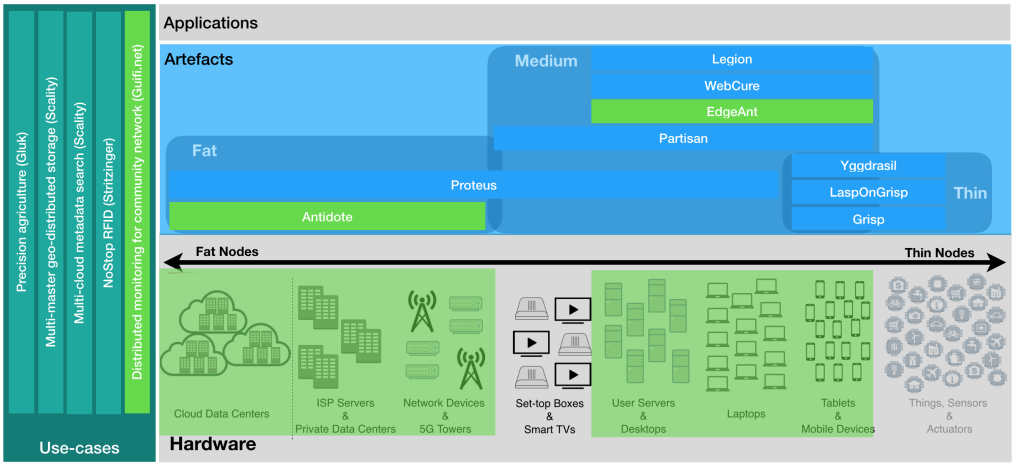


Figure 3.4.1: Distributed monitoring for community network (Guifi.net)

With regards to the artifacts developed in the LightKone project, as Figure 3.4.1, we currently use Antidote as a distributed storage service in the new implementation of the monitoring system. For the server to router assignment, decentralized read and write operation will be done on the storage service by each server in order to allow the coordination between the decision of each server. Yggdrasil and EdgeAnt have the potential to provide additional features to the use case implementation. For the storage of the monitoring date, for each router there will be monitoring data from several servers. This data will need to be merged or transformed into a single data stream or representation such that the data becomes useful for the needs of other network management services in Guifi.net.

The devices involved in the use case include commodity wireless routers and the heterogeneous servers, which monitor the routers and use the distributed storage service. The servers are heterogenous not only from the perspective of their their computing capability, but also from their connectivity, maintenance, ownership and operational policy. The devices thus represent features both from LightKone's heavy and light edge perspective. Note that monitoring monitoring service and the storage service mainly runs on the same hosts, and initially we do not foresee a separate infrastructure for the monitoring service and the storage service. We may however decide to use the set of more powerful servers for the storage service for the case of the monitoring data. The mentioned heterogeneous servers available in Guifi.net range from Single-Board-Computers such as the Raspberry Pi, to more powerful mini-PCs up to server-class desktop PCs.

3.4.2 Multi-cloud metadata search (Scality)

We first briefly summarize the context and system model of the use case, and we then present the current status of the system, and finally how we aim to use LightKone's artefacts to for implementing the use case.

(a) Context

Scality has implemented an open source framework (Zenko Multi-Cloud Controller) that enables applications to transparently store and access data on multiple public and private cloud storage systems using a single unified storage interface (the AWS S3 API). Applications can use Zenko to access multiple cloud storage systems, including Microsoft Azure Blob Storage, Amazon S3 and Scality RING with the same API, and can additionally define policy-based data replication and migration services among these clouds.

The focus of the use case is Zenko's capability to support federated metadata search across multiple cloud namespaces. This enables applications to retrieve data by performing queries on metadata attributes, such as file size, timestamp of last modification or user-defined tags and others, independent of the data location.

(b) System Model

The system model consists of a small number of geo-distributed cloud data centres, and a larger number of client devices (user servers & desktops, laptops). Some of the data centres fully replicate data, representing geo-replicated cloud storage systems, while others store disjoint datasets, representing different clouds storage systems (ex. DC1 & DC2 = AWS S3, DC3 & DC4 = Scality RING). An instance of Zenko is deployed on one of those data centres.

Clients perform reads and writes using the S3 API either through Zenko, who then forwards operations to the appropriate clouds (in-band operations), or by communicating directly with a backend cloud storage service (out-of-band operations). Clients can also perform metadata search queries through Zenko using an SQL-like interface.

In order to provide metadata search, Zenko captures and stores object metadata attributes in a database. This database is replicated within a data centre for fault-tolerance using a protocol such as Raft. MongoDB is used as this metadata database in the current implementation of Zenko. Metadata attributes are stored in MongoDB as JSON object and Zenko takes advantage of MongoDB's indexing and search capabilities to support metadata search.

For in-band write operations, metadata attributes are captured and stored in the metadata database. For out-of-band write operations, metadata attributes are eventually propagated to Zenko's metadata database using event notification mechanisms provided by the cloud services.

(c) Use-case implementation

The use case aims at introducing a geo-distributed metadata service as a replacement to the current more centralized approach which gathers and stores metadata attributes on a database placed on a single data centre.

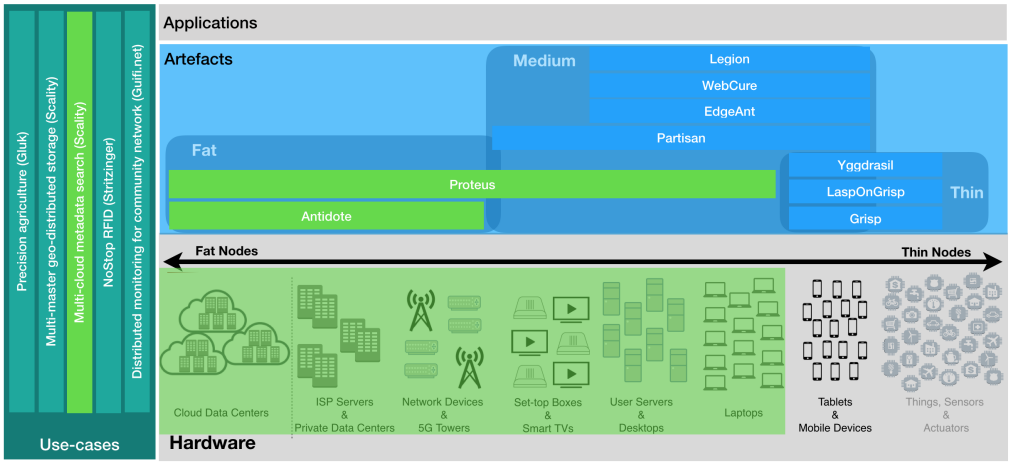


Figure 3.4.2: Multi-cloud metadata search (Scality)

For this, we will use case two LightKone artefacts: (1) Proteus and (2) Antidote, as convyed in Figure 3.4.2.

Proteus can be used for deploying a modular geo-distributed hierarchical network of microservices. Each microservice is responsible for receiving an input stream of data, optionally maintain a user-defined materialized view of the input steam, and provides a query-like service by performing data-flow computations on the input data and/or its materialized view. Proteus enables flexible placement of data and computations by allowing microservices to be placed on different parts of a geo-distributed system.

For this use we have implemented query processing microservices in Proteus. Each microservice receives write operations executed in a cloud storage system as an input stream, maintains a (partial) index on metadata attributes, and provides a metadata search service using this index.

These microservices are organised as a hierarchical network that implements a geo-distributed index. The index is partitioned, and index partitions are distributed among data centres. Metadata search queries are executed by performing distributed computations on this network: a given query is incrementally split in simpler sub-queries, and sub-queries are processed by different microservice components using their index partitions.

More specifically, we use Proteus by deploying query processing microservices as Docker containers on multiple data centres (and potentially even on client machines), and providing each microservice with a simple configuration describing its behaviour, its connections to neighbouring microservices within the network, and its connection with the underlying storage system. Proteus then handles the communication among microservices for performing distributed computations.

Microservices in Proteus use Antidote as a backend database for storing indexes as CRDTs. We are also currently investing using Antidote for providing causality and atomicity guarantees for search results (atomicity: a search query should either observe all the updates performed within a transaction, or none)

We aim to make use of Proteus' ability to enable flexible data and computation placement for allowing Zenko's metadata search service to make trade-offs between search latency, query result freshness and the storage overhead of maintaining indexes.

3.4.3 Multi-master geo-distributed storage (Scality)

(a) Context

Scality's object storage system (RING) supports "active-passive" geo-replication. Data are geo-replicated across multiple sites (data centres), but can be updated on only one of those sites. The other sites are read-only during normal operation, and serve as backups that are ready to take over in case of a failure of the active site. The goal of this use case, shown in Figure 3.4.3, is to support active-active geo-replication in Scality's object storage system, where data on multiple sites will be updated simultaneously and changes will be merged deterministically.

(b) Data and system model

The data model is that of object storage: the system stores a set of objects, organised in buckets which form a flat namespace. Each object consists of a key that uniquely identifies the object within a bucket, a blob of uninterpreted data, and a set of metadata attributes, such as content size, creation timestamp and user-defined tags. Buckets also contain similar metadata attributes. More importantly each bucket maintains a primary index with the keys of the objects which it contains. Data are immutable, modifying the data creates a new version of the object, while metadata attributes can be updated. Scality's storage system performs separation of data and metadata. Data are stored in Scality's RING storage platform while metadata are stored in a separate metadata database.

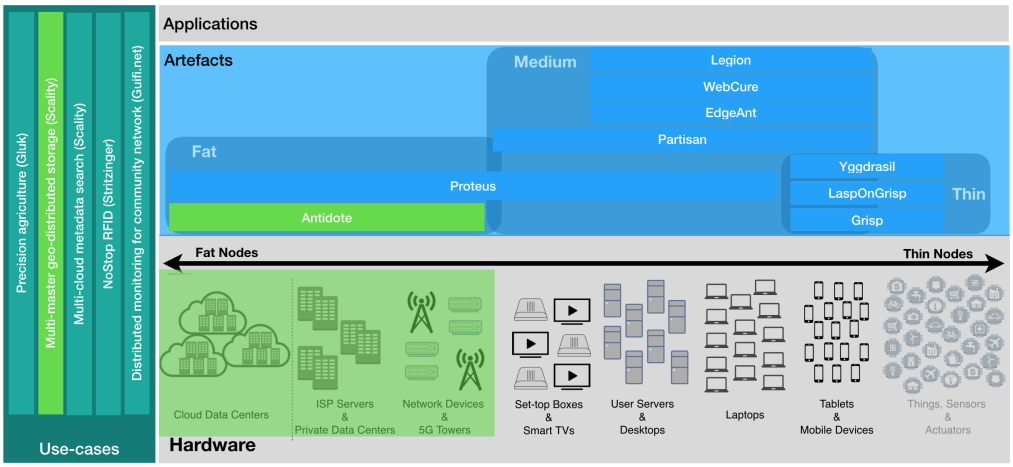


Figure 3.4.3: Multi-master geo-distributed storage (Scality)

(c) Use-case implementation

The key idea of this use case is to use Antidote as a metadata database for Scality's object storage system (as shown in Figure 3.4.3). An Antidote data centre will be deployed on each site. Metadata attributes will be stored in Antidote as CRDTs, and Antidote will handle the geo-replication and convergence of concurrent updates. Data will still be stored in the Scality RING and replicated using the existing mechanism.

3.4.4 NoStop RFID (Stritzinger)

(a) Context

Prior to the project we have developed several RFID subsystems for Boschrexroth some of them are widely used in production at industrial automation customers of Boschrexroth and others have been used as prototype in research and innovation demonstrators. Usually these RFID systems communicated with a reusable TAG mounted on workpiece carriers in manufacturing tracking and guiding the production steps for a single workpiece in manufacturing per tag. Since the used RFID radio systems are short ranged they can also provide reliable identification of which workpiece is entering a processing station e.g.

These legacy systems are implementing a local cache for the RFID content of the tags before the antenna. These local caches are mainly to provide optimized abstraction of the block structure of most of modern tags and some limited drive by behavior of automatically reading user configurable parts of the tag content which then can be perused later by a local manufacturing process the reader is connected to.

The legacy model in practice very often needs to stop the workpiece carrier to either read more data and always to reliably write data (which requires a read, modify, write, read for verify sequence), throughput and utilisation of a factory is reduced by these waiting times which either occur before and after a processing station or even in the station itself. In any case this results in unnecessary pauses where the processing station is idle.

The new system we are prototyping in LightKone will replace these localized caches by a fully distributed cache over the networked RFID readers of a assembly line. When implementing this we need to take care to keep the semantics of the preexisting system from the view of a processing station accessing a single reader before. We keep this single connection from processing station to RFID reader network node (while now being open to add redundant connections to other nodes for fault tolerance in the future).

(b) Data and system model

We plan to implement this distributed cache with the semantic of an array of register CRDTs. One for each byte in the RFID content, containing the value if known and metadata for caching functionality. The semantics for concurrent writes would be “last writer wins” with the additional twist that the write ordering is determined by the physical path of the RFID tag through the manufacturing system. In order to allow this and provide write operations to a process station after the tag already left the reader position (we don’t stop anymore) we add to these register writes a version field which is stored on the RFID tag and incremented every time it is accessed for writing at a RFID antenna. These version fields can be used to order the writes in the distributed cache in the semantically correct ordering no matter in which temporal order they have been processed and distributed in the Network.

In order to associate the correct cache content with a certain tag all these arrays of registers will be stored in a distributed Key-Value (KV) store. The keys in this KV store are the unique id that all modern RFID tags have built in. The values are the arrays of CRDT registers as described above.

For anti entropy process in this distributed KV store with CRDT values we plan to use a lightweight gossip protocol implementation. From other parts of the future system

we will have physical network topology information which allows us to identify our neighbors and set up the gossip protocol in a way that no redundant information is sent over the same physical network link twice, reducing the network utilization.

(c) Use-case implementation

We have ported our GRiSP platform for embedded Erlang systems to Boschrexroths RFID reader hardware, directly using this LightKone technology as basis of the implementation.

For efficiency and space reasons we will implement the above described CRDT semantics of a array of register CRDTs (one for each RFID tag byte or possibly even bit) with a more efficient implementation of the whole array but with the same semantics. Therefore we probably can't use a library of predefined CRDT data-types. We will work together with researchers in the project to define a possible generic interface (Erlang behaviour) to a underlying complex CRDT implementation which can also be used in the CRDT libraries of the project.

This makes it possible to implement the lightweight distributed KV store in a generic manner so it could be used with CRDT libraries or application provided CRDTs. After researching the available options in the project we have identified a possible candidate for a KV store at partner INESC called Dante KV which while it has been developed inside the LightKone project can currently not be part of the reference architecture because the group disagrees if it can be mentioned in this version of the document.

Either on basis of the prior artefact or from scratch we will implement this lightweight KV store with gossip anti-entropy as part of our use case, which will then join the other technologies in the LightKone Reference Architecture.

3.4.5 Precision agriculture (Gluk)

(a) Context

The Self Sufficient precision agriculture management for irrigation use case will be applied for irrigation management in Subsurface Drop Irrigation method (citrus trees cultivation - but can be applied in every farming activity indoor or outdoor).

In the current approach, the water is pumped out from the well (or other water source) and transmitted to the polymer tubes. The water usually is used to irrigate multiple farms. However, as every farm has different characteristics (soil, area, etc.), the irrigation should be adapted taking account these parameters. For example, some parcel of land may need more water while other parcels need less water.

In order to avoid under-watering, the farmers usually irrigate more time than it is necessary. This raises the issues of water waste, energy waste (electricity for the pump), and drainage problems (since the same time many farmers irrigating and the water in the underground water dump is not enough for everyone). In the current situation we don't have a clue which part of the farm is either over-watered or under-watered. We can see only afterwards when the tree is not full of products or its leaves are yellow, etc. (again empirical and observation methods). A non-proper irrigation could affect 25-30% of the annual production.

(b) System model and Requirements

Using the Self-Sufficient management system, the farm will be divided in clusters and in every or middle of the tube will be installed a smart node containing the management unit, sensors and actuators. In that way the farmer could divide into zones his farms and when a zone is sufficient irrigated (retrieved value from the sensor) the actuators will stop the water flow into specific parts of the tubes. The rest of the farm that still needs water will continue being irrigated.

The core management ability must be completely autonomous (no need for PC or cloud control) and as low-cost as possible (again, no need for PC or cloud connectivity, which can be expensive), and for this it should run on the sensor array itself. The system must be able to be installed by the farmer without any configuration capabilities from his side (zero config). Additional management abilities can be added,which will cost extra, but they are not essential for the correct operation of the system.

Requirements for the management software The basic management should be done by the sensor array itself. Higher-level management goals can be added by external systems, such as PCs or cloud tools, but such external systems cannot be guaranteed to be connected to the sensor array. Also, we would like the system to be as low-cost as possible: the most essential management should be done on the sensor array itself, without any external costs. This gives modularity for the farmer: he pays only for what he needs, and Internet connectivity is not needed for basic management abilities.

Requirements on the sensor array In order to achieve this, the requirements on the sensor array are that there should be (1) basic computation ability in the sensor nodes, and (2) basic communication ability between sensor nodes (for example, Wifi or Zigbee), with normal reliability of these nodes as provided by off-the-shelf hardware.

Given these requirements, the software we develop using LightKone technology should be able to perform reliable basic management (24/7) despite problems in the sensor array (nodes going down, communication being unreliable).

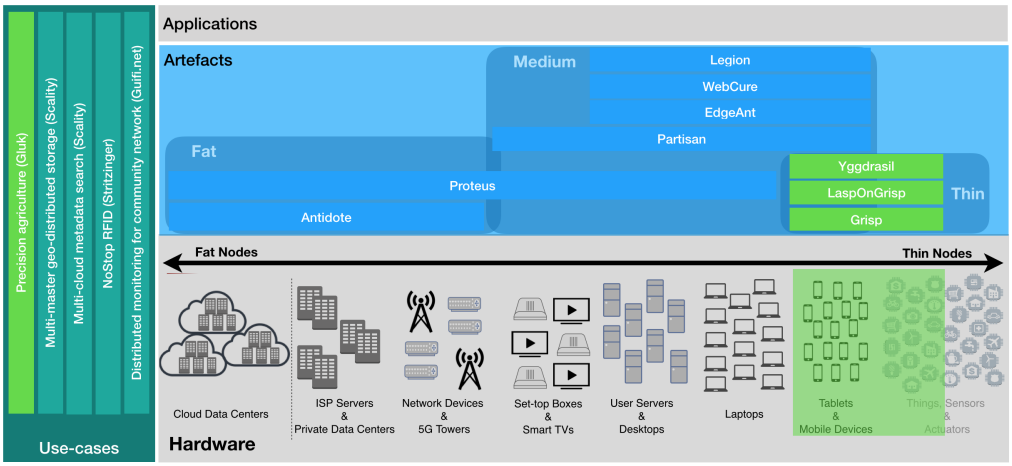


Figure 3.4.4: Precision agriculture (Gluk)

(c) Use-Case Implementation

We will use LightKone technology to provide reliable computation and communication ability despite unreliable nodes and communication. As shown in Figure 3.4.4, we will experiment a Proof of concept using LasponGRiSP and possibly Yggdrasil. Lasp provides a reliable replicated key/value store that runs with very little computational resources, on top of a communication layer, Partisan, that ensures reliable communication despite highly unreliable connectivity (using hybrid gossip). Basic connectivity provided by Yggdrasil underneath Partisan. We will extend Lasp with a simple task model that stores the management software in the Lasp store itself (which is possible because of higher-order nature of Erlang), and performs periodic computations, storing results in the Lasp store. GRiSP provides native Erlang functionality running with low power, with basic processor power and memory and wireless connectivity. GRiSP also provides Pmod sensor interfacing to provide the sensor and actuator abilities.

GRiSP nodes can be powered by solar batteries. 100% uptime is not required because of the Lasp redundancy. Occasional problems in individual nodes are solvable by periodic reboot of individual nodes. This will not hinder overall system operation because Lasp replication and Partisan hybrid gossip are designed to survive such problems.

Management policy control is provided by a connection to the sensor array, either by PC or cloud, which the farmer can do at any time. This connection does not need to be continuous or reliable. The management will continue to work even if the connection is not done for several days or more.

3.5 Edge/Fog System Models and Terminology

In this section we define Heavy, Light, and Hybrid edge, describing their properties and mapping these categories to LightKone use-cases. This terminology overlaps and complements SOTA definitions as we discuss in the next section, but it is more descriptive to application-layer data patterns, being the focus of LightKone. We start by presenting some definitions.

3.5.1 Taxonomy and Definitions

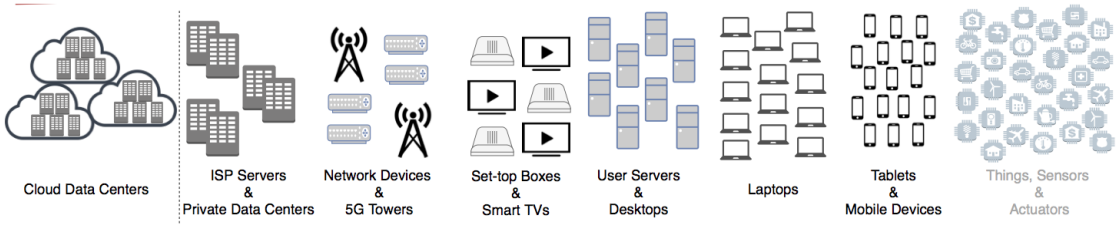


Figure 3.5.1: Edge devices spectrum.

We note the existence of a wide variety of edge devices with diverse characteristics and capacities. A fine-grained classification as in Figure 3.5.1 is interesting to understand the application in detail, but our observation across the use-cases leads to the following three main categories considering available storage, computation, memory, network bandwidth, power, etc.:

- **Fat node:** an edge/fog node or device having considerable capacity. E.g, a commodity server.
- **Medium node:** an edge/fog node or device having moderate capacity. E.g., a computing router or gateway.
- **Thin node:** an edge/fog node or device having constrained capacity. E.g., an IoT device or mote.

On the other hand, we also note the importance of data-flow patterns across use-cases:

- **Unidirectional dataflow:** data mostly (an order of magnitude) flows in one direction through the network (e.g., pushing aggregated data from sensors to processing machines).
- **Bidirectional dataflow:** data flows in two opposite directions in the network (e.g., pushing sensed data to processing machines, and processed data to actuators).
- **Omnidirectional dataflow:** data flows in multiple directions through the network, i.e., in ad-hoc but possibly known manner (as in Gossip, Mesh, etc.).

Furthermore, the type and role of nodes in an edge network can change the system properties, and thus we define the composition of an edge network as follows:

- **Symmetric edge network:** nodes mostly have the same role and capacities (e.g.,two peers in IoT network)
- **Asymmetric edge network:** nodes mostly have different roles and maybe different capacities (e.g., a client and server)

3.5.2 Heavy Edge

In heavy edge, the communication mostly occurs north-south, mainly between fat and medium/thin devices. This can be seen as a distributed variant of classical server/client service at the edge, and thus the network is asymmetric. Fat nodes often play a major role in storage and computation to serve the requests issued by medium/thin nodes, and hence, the dataflow is unidirectional (in aggregation use-cases) and bi-directional (in case of control tasks and presence of "actuators"). In addition, fat nodes are often static in terms of location and membership, which helps assuming some properties to implement consensus protocols and transactions. Scality's use-case on distributed highly available indexing in storage is an example of heavy edge. In this use-case, clients delegate the bulk of the work to fat edge nodes. In particular, AntidoteDB can be used as a heavy edge database for such use-case.

3.5.3 Light Edge

In light edge, the communication is mostly lateral (south-west) between thin/medium nodes. The model is often Peer-to-Peer (P2P) communication style where peers almost have equal roles and rights; therefore, the network is basically symmetric and communication can be omnidirectional in an ad-hoc pattern style (usually built on Mesh networks,

DAGs, etc). Furthermore, such networks are often implemented over hostile wireless networks (WIFI, Zigbee, Bluetooth) and thus, there is high rate of churn in addition to dynamic membership and locality (e.g, in mobile networks, robots, VANETs, etc.). This suggests using data and communication abstractions that are more robust in face of such challenges as in state-based CRDTs and hybrid-gossip protocols, respectively. For instance, Stritzinger's use-case on RFID-powered conveyors can experience transmission problems due the nature of ad- hoc wireless protocols between conveyor stations and the constrained devices (RFIDs) on products under fabrication, and allows for dataflow in all directions (across different stations and products). This makes it more convenient to use a gossip-based communication protocol (instead of a causal delivery middleware) together with state-based CRDTs being tolerant to duplications.

3.5.4 Hybrid Edge

Hybrid edge comprises systems where there is no dominance of the heavy or light edge and thus the communication includes north-south and east-west patterns. This refers to use-cases that can cover different patterns and settings and thus exhibit both symmetric and asymmetric networks, server/client as well as P2P communication, etc. We give two concrete use-cases to exemplify the need for such category. The first is Guifi's use-case on community networks in which people can offer and run services on their own nodes and publish to the network, and special nodes can monitor the network's state or play the role of data stores. The system can be coordinated through heavy edge, e.g., Antidote, whereas there is no direct interaction between nodes (i.e., the traffic goes through the heavy edge system). However, this model can be extended to have the nodes directly talk to each other in a fully decentralized way, e.g., neighbors can publish services to each other without passing through the heavy edge system (e.g., Antidote in this case). At the same time, the monitoring service of Guifi will likely be semi-decentralized to reduce the overhead of decision making and provide the necessary availability and resilience.

Another very common use-case is the IoT time-series based use-cases as Gluk's precision irrigation control system. In this use-case, IoT devices (sensors and meters) are used to detect the physical environmental state (e.g., humidity, salinity, temperature, illumination, etc.); the corresponding data is aggregated and then pushed to more powerful machines (e.g., cloud or heavy edge) to be processed. In the cases where the system gets large, thin/medium nodes (sensors and gateways) can interact without the need for the cloud center but possibly with intermediate upper fog layers. Thus local aggregation and basic computations occur at lower layers (hence the light edge part) before delegating the processing to more powerful machines at an upper layer (hence the heavy edge part).

3.6 Related Work

We now discuss how the LiRA relates to proposals that address the design of generic and specific (mostly IoT) edge computing solutions.

3.6.1 OpenFog RA

The OpenFog Reference Architecture (ORA) is a medium- to high-level view of system architectures for fog nodes and networks. It is the result of a broad collaborative effort of its independently-run open membership ecosystem of industry, technology and university/research leaders. It was created to help business leaders, software developers, silicon architects and system designers create and maintain the hardware, software and system elements necessary for fog computing. It enables fog-cloud and fog-fog interfaces.

ORA can be considered a standard architecture for fog computing for two reasons. First, it is a joint work of a broad consortium of of leading technology bodies from industry and academia with diverse expertise in the entire hardware and software stack of cloud/fog/edge computing; and second, the architecture is broad enough to touch the majority of aspects and concerns in fog ecosystem considering multiple views, different perspectives, as well as general desired properties. (https://www.openfogconsortium.org/wp-content/uploads/OpenFog-Reference-Architecture-Executive-Summary.pdf)

The LightKone Reference Architecture (LiRA) is complaint, but complementary, to the OpenFog Reference Architecture (ORA). It is complaint as it often follows the same terminology, general software architectural of a fog node, and targets the common themes (a.k.a., ”pillars” in ORA) like, availability, autonomy, scalability, reliability, etc. However, it is complementary as LiRA emphasizes more the application level layer, specially the distributed data management and communication techniques. This sometimes leads to defining new terminology that is needed to classify application data and communication patterns that was not addressed in ORA (in the current version [**?**]). We explain the relation to ORA in more details in the following.

(a) Distributed Data Management

ORA touches upon data management at a very high level through mainly emphasizing the need to support various types of durable persistence, in-memory caches, SQL and NoSQL databases, but other forms of durable storage should be considered, such as in-memory databases , etc. In particular, LiRA does not support lateral data sharing across edge devices of the same level as LiRA does. In ORA, data sharing is only done through a higher level node in the hierarchy of the fog network. This is clearly described in the Visual Security and Surveillance Scenario use-case section. Furthermore, ORA does not tackle with application-specific data requirements and invariants as LiRA provides data consistency, convergence, invariants, etc. ORA only mentions some required features at the application-level like data encoding, encryption, and support for structured/unstructured data.

(b) Fog vs Edge Computing

The LightKone Grant Agreement (LKGA, written end of 2016) document predates ORA (published in 2017) and therefore their terminology is slightly different. What ORA calls Fog Computing, which spans from data-centric cloud to edge devices (or north to south), is close to what LiRA calls “Heavy Edge”. What ORA calls Edge Computing, focusing on what happens on and between the edge devices themselves (or east to west), is close to “Light Edge” in LiRA. Such a divergence in terminology is not uncommon in fast-moving fields. (For consistency with the other LightKone documents, we use

herein the terminology from LKGA, and occasionally refer to the OpenFog terminology because it is in wide use.) Nevertheless, the notion of Heavy Edge and Light Edge are broader, as defined in the previous section, considering the communication patterns, node capacity, and dataflow of edge applications, which was not sufficiently addressed in ORA. In addition, LiRA defines Hybrid Edge as a mixed model of both Heavy Edge and Light Edge, in which communication and data flow occurs east to west and north to south, thus allowing data sharing between nodes on the same level even if the upper layer is unavailable.

(c) Pillars or themes

LiRA also shares with the ORA the importance of Fog/Edge functional and nonfunctional properties, called Pillars or themes in ORA. LiRA however seeks availability and autonomy from an application-layer data perspective, complementary to ORA. Specifically, ORA often demonstrates availability at the management and orchestration levels, e.g., redundant nodes, MTTR, redundant configurations, etc., and autonomy as operating when the cloud is unavailable. However, application layer requirements, like consistency, can greatly prohibit availability even if the infrastructure is highly available.

3.6.2 EdgeX

EdgeX Foundry (https://www.edgexfoundry.org) is a Linux Foundation project that hosts the development of the EdgeX software platform. EdgeX was announced in April 2017 as new Linux Foundation project supported by 50 companies. EdgeX is described as a vendor-neutral common open framework for IoT edge computing, targeting specifically industrial IoT. EdgeX was originally developed by DELL in Java before it was released as an open source in a Linux Foundation project. In recent releases the original Java code became more and more replaced by new implementations in Golang (https://github.com/edgexfoundry/), making the EdgeX software more lightweight and therefore it can become suitable to also run on devices with lower capacities.

Architecture. EdgeX is organized in a layered architecture of four layers named *Export Services*, *Supporting Services*, *Core Services* and *Device Services*, along with two vertical layers addressing security and system management. This architecture interfaces on the south side (Device Services layer) to the IoT devices. Over this interface it communicates with the physical IoT devices and receives data. On the north side (Export Services layer) it interfaces to a Cloud system. The Cloud system collects, stores, aggregates and analyzes the data. EdgeX therefore enables data to be sent over it from south to north and the other way around, while within EdgeX, by the Support Services layer, some processing (analytics) can be done, as to the services available in the support service layer.

Data Storage. There is data storage in the EdgeX platform provided by the Core Services layer. Data from IoT devices is stored by the *Core Data* component, a component described as a a persistence repository for data collected from the south side objects. Data about the EdgeX configuration is stored by the *Metadata* component, a component

described as a repository service for metadata about the objects that are connected to EdgeX.

Hardware. EdgeX targets to be deployable on heterogeneous hardware, from powerful gateway servers to low-capacity devices such as the Raspberry Pi.

Services. EdgeX is microservice based, where each microservice is provisioned as a Docker image. As such, EdgeX is very modular and flexible with regards to the composed services, which can make it adaptive to specific conditions.

Deployment. The current stable EdgeX implementation seem to be target for being deployed on a single device, but not on distributed hardware. Given that on the north side EdgeX interfaces to a cloud system, it can be argued that this cloud system provides the needed resource elasticity to respond to different workload situations of edge applications. However, the developments for the recent releases of EdgeX also include efforts to target for a distributed deployment of microservices. It can also be seen as a limitation that the elasticity of the services components of EdgeX itself are bound to the computational resources of the hosting device.

Opportunities for LightKone. EdgeX is a recent platform (with public launch in April 2017, at the time of writing of this text this was not much more than 1.5 years ago). News and events on EdgeX, frequent releases and industry support however indicate an important effort behind EdgeX and the potential to become a key open source platform for IoT applications. A demo of EdgeX at IoTSolutions in October 2018 showed the pursue of EdgeX to become production ready. At current time, however, only prototypes seem to be ready and there is not a yet any clear operational commercial cases running EdgeX reported. LightKone persistent storage technology may contribute to the persistent data storage services from the Core Services layer to become distributed. Such an extension could complement EdgeX's data storage services with additional properties. This could fit into the releases of EdgeX planned for 2019, which aim to include a better support for the distribution of service components to east and west (https://wiki.edgexfoundry.org/display/FA/Fuji+Release). On the other hand, the roadmap of EdgeX needs to be further understood. EdgeX seems to aim for reaching production readiness at the shortest possible time frame and to fit to the audience of IoT applications in general. It can be expected that the general open source release of EdgeX will integrate the capabilities needed for general IoT application requirements, while for more specific and challenging IoT applications, the extra support for using EdgeX will be given through commercial value-added services of the partners of the EdgeX ecosystem.

3.6.3 ECC RA

The ECC Reference Architecture (called ECC RA in the following) is a model-driven reference architecture that supports autonomy and collaboration of edge devices. It is defined jointly by the Edge Computing Consortium (ECC) and the Alliance of Industrial Internet (AII), where both consortia are spearheaded by major Chinese Internet companies and the Chinese government. The ECC RA 2.0 document was published in Nov.

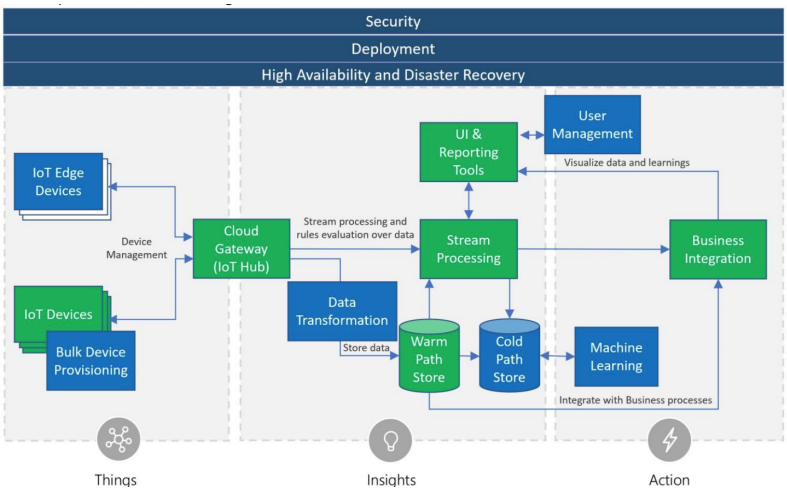


Figure 3.6.1: Azure IoT RA.

2017. ECC RA's model support includes frameworks for real-time computing, limited-resource computing, full-featured gateways, and full distributed system supports. ECC defines edge computing as an open distributed platform running close to edge devices (called "things") and data sources, and subject to hostile environmental conditions. The ECC architecture is layered, with edge computing nodes (ECN) at the bottom, then a connectivity/computing/storage layer (CCF), then a service layer, and finally a set of so-called smart services. Crosscutting concerns over all layers consist of security (including identity and authentication), lifecycle management, and emergency response management. The service layer (called "fabric" in the documentation) supports services to add specific functionality. There is specific support for time series database mentioned in the architecture, since computation with time series is a common application in edge computing. Deployment targets a hierarchical infrastructure with edge devices, gateways, and cloud services as the main layers, similar to the generic infrastructure of D2.1. The programming model is based on the CCF, which provides operations for connectivity, computing, and storage. All services can use this layer. An eventual integration of LiRA components into ECC could focus on extensions to the CCF, or services that effectively provide extensions to the CCF.

3.6.4 Azure IoT RA

Microsoft Azure IoT Reference Architecture (Azure IoT RA) provides the recommended architecture and implementation technology choices for building Azure IoT solutions. Azure IoT solutions are composed of *Things* (or devices), which generate data or events that are sent to the cloud for storage and processing to generate *Insights*, which are used to generate Actions.

According to the Azure IoT RA, presented in Figure 3.6.1, an IoT application consists of the following subsystems:

1. devices that send data to and receive data from the cloud using different communication options. Optionally, an IoT Edge Device may execute some data processing at the edge;

2. a cloud gateway service, or hub, to receive data and manage devices;

3. stream processors that consume data and store it in the storage;

4. storage to store data received from devices or the results from stream processors. The business processes consume the results from stream processors and from data stored in the storage;

5. a user interface to visualize data and learning, and to simplify management.

For each of these subsystems, Azure IoT RA proposes multiple options, that can be used depending on the requirements of the solution being proposed.

When compared to LiRA, Azure IoT RA misses the Light-edge layer. The reason for this is that the Azure IoT RA focus only on IoT applications and it leverages highly on Azure cloud infrastructure. The goals of the subsystems on Azure Things are similar to the goals of the artefacts we have been developing for the LiRA thin layer: collect data from devices, process them and propagate them to be sent to some other system - while in Azure, it is assumed that data is always propagated to the cloud, in LiRa we assume that data can be propagated to some Light-edge subsystem. On the cloud (Fat layer in LiRA), we have been focusing mainly on storage and searching, while Azure RA focus also on stream processing of received data. Given the large number of data processing systems currently available, we assume that applications will rely on third party systems to process received data, as discussed before.

3.6.5 Amazon Greengrass RA

AWS IoT Greengrass is an AWS solution for extending AWS to edge devices to allow data to be processed at the edge. While using Greengrass, applications still use the cloud for management, analytics, and durable storage. At the edge, devices execute predictions based on machine learning models, synchronize data with the cloud and communicate with other devices. At the cloud, AWS services for storage, analytics and data management can be used. For each type of services, different AWS solutions are available and can be used depending on the applications requirements.

The general architecture of Greengrass is similar to the one presented in the Azure IoT RA. Thus, the discussion in the previous section regarding the comparison with LiRA also applies to Greengrass. Still, some aspects are worth being highlighted. First, one of the Greengrass goals is to support offline operation. The artefacts we have been developing in the context of LightKone also focus on this goal. Second, Greengrass allows to use languages and programming models used in AWS cloud services (e.g. Lambda functions). The LightKone project also tries to develop solutions that can be used in all layers of the LightKone architecture, notably CRDTs. We are also specifying an unified semantics for the computations that execute in the different artefacts being developed in the context of LightKone.

Table 3.2.1: **Sumamry of the main LiRA artefacts.**

Artefact	Description	Previous SOTA	Contribution	Read more
AntidoteDB	A highly available geo-distributed database	Geo-replicated databases with different consistency semantics, typically either weaker (EC) or stronger (Serializability within shards)	Causal transactions + CRDTs	D6.1
WebCure	Client-side data replication for web applications using AntidoteDB as backendTODO: Annette	Read-only caches / roll back on updates on conflict	Simplified programming model with conflict resolution on CRDTs	D6.2
Legion	A framework for extending web applications to the edge, by running code in the client devices that interact directly.	Systems that support disconnected operation, but no peer-to-peer synchronization; Mobile systems that support peer-to-peer interaction, but that are not designed to support web applications.	Simple programming model for extending web application with peer-to-peer synchronization.Big delta CRDTs.Model for interacting with cloud services.Security mechanisms.	D5.1
EdgeAnt	A consistent, mutable cache at the edge. Data is backed up in Antidote. EdgeAnt supports the same API as Antidote, and guarantees the same TCC+ consistency. A cache can transparently disconnect and reconnect to any data centre. Ongoing work: (i) A client has the option to place any individual computation, either at the edge or in a data centre; both guarantee the same consistent view of data. (ii) Co-located EdgeAnt clients can collaborate in a group, even disconnected from the infrastructure, and can migrate between groups.	Edge caching for immutable data; or non-AP "sticky" or ad-hoc caches with ill-defined guarantees	Consistent, mutable AP cache. Uniform (DC to edge) AP guarantees Client can migrate Place computation @edge or @centre P2P group communication Client can change groups	D6.2
Yggdrasil	Framework for designing distributed protocols for ad-hoc networking.	Frameworks for developing distributed protocol, but no specific one for wireless ad-hoc networking.Multiple protocols for ad-hoc networking.	Simple programming model for defining new protocol, hiding the complexity of configuring wireless radios and exchanging messages among multiple communication.parties.	D5.1
Proteus	A geo-distributed framework for analytics computations on federated data stores. Proteus maintains materialized views and performs stateful data-flow computation. Admins place computation and data according to SLA considerations.	Apache Spark,Distributed search for federated clouds, Federated query processing on linked data,Lasp??	Bidirectional data-flow computations using materialized views stored as CRDTs. Modular distributed architecture that enables flexible data and computation placement in geo-distributed systems.	D6.2
Grisp	A Unikernel approach running the Erlang VM directly on smaller hardware without intervening operating system level. There is a software stack that allows for mixed critical systems with hard and soft real-time parts. A evaluation and development board for this was developed outside the project and provided to partners.	Running Erlang on Embedded Linux like operating systems. Soft real-time only.	Erlang on smaller IoT devices which wouldn't be able to run Linux. Erlang as part of mixed critical systems. Preparation for allowing hard real-time Erlang processes.	D5: Chapter 3.4
LaspOnGrisp	Reliable key/value store running on network of Grisp boards, allowing applications to run directly on the sensor boards. Reliable data storage based on CRDTs; reliable communication based on hybrid gossip (Partisan).	SOTA edge applications do not run on sensor networks, but on gateways that manage these networks.	Resilient data storage and resilient communication directly on sensor networks.	D4.2

Chapter 4

Plan and Progress

The purpose of this chapter is to provide the required data and communication abstractions that constitute the building blocks of the edge computing runtime. The main significant data abstraction is based on the idea of Conflict-free Replicated DataTypes (CRDTs) [84] previously developed in SyncFree FP7 project[1], a predecessor of LightKone. Since they are proven to achieve high availability and convergence in loosely connected network, we adopted the CRDT approach for the edge as well, and we tried to improve and extend them in many directions to fill the gaps on the edge. We have also focused on adapting the needed communication abstractions for this approach by adopting hybrid-gossip protocols (like Partisan) and reliable causal broadcast (both described in later sections). As stability is the major challenge at the edge, we opted to dedicate a separate section in which we introduce several techniques and protocols to boost scalability. Although these approaches can fit in other sections, we follow this structure to show the emphasis on scalability.

4.1 Plan and Milestones

We first present the plan we followed during the first year to develop the initial data and communication abstractions of generic edge runtime, and the prospective plans for the rest of the project. The plan is often consistent with that presented in the original proposal (unless explicitly mentioned otherwise). We also took into consideration the support for generic edge computing runtimes considering the LightKone use cases and pushing the envelope further to explore and understand the potential of LightKone technology beyond our use cases. We believe that is important to create more impact and innovate.

4.1.1 Plan followed in Year 1 (Y1)

In the first year, we aimed at providing the initial data and communication abstractions and protocols to build a generic edge computing runtime. These abstraction and protocols will be eventually packaged as software components that can be used in edge runtime within LightKone, as in the other work packages, and importantly outside LightKone. As explained in the previous chapter, the SOTA of edge/fog computing did not sufficiently address distributed data management in a way to allow sharing data efficiently across fog

[1] https://pages.lip6.fr/syncfree/

layers and within the same layer. In LightKone, we decided to build on the CRDTs [84] invention the academic partners developed in the past EU FP7 project Syncfree [34]. CRDTs have proven to improve the availability of geo-replicated systems through automatic conflict resolution in relaxed consistency model, but they were not feasible to edge networks with constrained resources, big number of edge nodes, and hostile network. In WP3, we planned to fill this gap by mainly extending CRDTs to edge/fog, through addressing the application data layer and communication layer. We started by addressing the general edge-tailored features like supporting more datatypes and improving their efficiency. Meanwhile, the use case semantics started to appear and we continued addressing edge application layer semantics like convergence, causal consistency, strong eventual consistency. We also started extending these CRDTs to support edge computing applications through improving their efficiency (reduce meta-data storage and dissemination), scalability (using partial replication), and resilience. This targeted the datatypes themselves as well as their underlying communication layer like causal broadcast and group membership protocol. Finally, we started preparing the security analysis for the use case and suggesting off-the-shelf.

4.1.2 Plan for the first half of Year 2

In the following six months of Year 2, we aim to continue our work through addressing the application semantics already started in Y1. In particular, we aim to continue improving the efficiency of the different CRDT models through optimizing the datatypes, mainly the pure operation-based and delta-state CRDTs, as well as their underlying communication layer. At the communication layer, study the impact of the causal broadcast on concurrent applications and identify potential implementation pitfalls. We will also evaluate the performance of the state exchange in the delta-data model and assess the performance of Partisan library. We also aim to continue the work started on partial replication as described later in this chapter and to provide a security analysis of the use case threat models.

4.1.3 Plan for the second half of the project

Considering the comments of the reviewers of the European Commission, the plan for the second half of the project is subject to an update to design the Reference Architecture (LiRA) that governs the entire project. This work is part of WP3, and is presented in this deliverable D3.1. Our plan by the end of Year 3 is to improve performance and resilience of the developed protocols. Specifically, we aim to increase the scalability of CRDTs (to the number of edge nodes) to one order of magnitude. We will also address the issue of dynamic networks and churn. Finally, we will explore the potential of Hybrid edge applications defined in the previous chapter building on the outcome of WP5 and WP6 on Light and Heavy edge. At the end of the project, we will delivered packaged libraries and protocols that can be used beyond LightKone. We provide a more detailed plan in the next deliverable D3.2.

4.2 Data Abstractions at the Edge

LightKone adopts the use of Conflict-free Replicated Data Types (CRDTs) to improve the availabilty and reslience of edge computing systems. Although proven successful in geo-replication, the current state of the art of CRDTs cannot answer the needs of the edge for several reasons attributed to the hostile edge networks and constrained devices, e.g., efficiency, scalability, heteroginity, etc. In this report, we focus on efficiency, and we aim at further progress in future reports. In particular, we present optimizations for several variants of state-based and op-based CRDT models through reducing the meta-data shipped over the network and stored in devices. Some of these contribtions are practical, and involved hands-on optimizations for CRDT implementations embedded in LightKone arifacts, as in Antidote DB (presented in detail in WP6). In addition, we tried to fill the gap of missing datatype sepcficiations by providing a portfolio for various variants of counters, registers, sets, maps; and support important and challenging operations as "reset". Since CRDTs are key in this project, we opt to start with a convenient comprehensive background for the convenience of the reader.

4.2.1 CRDTs: state-of-the-art and beyond

Conflict-free Replicated Data Types (CRDTs) are data abstraction tools that can be particularly helpful in edge computing scenarios. In the following summary we highlight some of their properties and key research findings. A more complete coverage of the state-of-the-art, as of 2018, will be available in an upcoming book chapter [84]. Details on the particular improvements of CRDT models, both state-based and operation-based, are covered in Sections 4.2.2 and 4.2.3.

(a) Overview

Internet-scale distributed systems often replicate data at multiple geographic locations to provide low latency and high availability, despite outages and network failures. To this end, these systems must accept updates at any replica, and propagate these updates asynchronously to the other replicas. This approach allows replicas to temporarily diverge and requires a mechanism for merging concurrent updates into a common state. CRDTs provide a principled approach to address this problem.

As any abstract data type, a CRDT implements some given functionality and exposes a well defined interface. Applications interact with the CRDT only through this interface. As CRDTs are specially designed to be replicated and to allow uncoordinated updates, a key aspect of a CRDT is its semantics in the presence of concurrency. The concurrency semantics defines what is the behavior of the object in the presence of concurrent updates, defining the state of the object for any given set of received updates.

(b) Concurrency semantics

The operations defined in a data type may intrinsically commute or not. Consider for instance a Counter data type, a shared integer that supports increment and decrement operations. As these operations commute (i.e., executing them in any order yields the same result), the Counter data type naturally converges towards the expected result. In this case, it is natural that the state of a CRDT object reflects all executed operations.

Unfortunately, for most data types, this is not the case and several concurrency semantics are reasonable, with different semantics being suitable for different applications. For instance, consider a shared memory cell supporting the assignment operation. If the initial value is 0, the correct outcome for concurrently assigning 1 and 2 is not well defined.

When defining the concurrency semantics, an important concept that is often used is that of the *happens-before* relation [61]. In a distributed system, an event e_1 *happened-before* an event e_2, $e_1 \prec e_2$, iff: (i) e_1 occurred before e_2 in the same process; or (ii) e_1 is the event of sending message m, and e_2 is the event of receiving that message; or (iii) there exists an event e such that $e_1 \prec e$ and $e \prec e_2$. When applied to CRDTs, we can say that an update u_1 *happened-before* an update u_2 iff the effects of u_1 had been applied in the replica where u_2 was executed initially.

As an example, if an event was "*Alice reserved the meeting room*", it is relevant to know if that was known when "*Bob reserved the meeting room*" to determine if Alice should be given priority or if two users concurrently tried to reserve the same room.

For instance, let us use *happened-before* to define the semantics of the *add-wins* set (also known as observed-remove set, OR-set [92]). Intuitively, in the *add-wins* semantics, in the presence of two operations that do not commute, a concurrent add and remove of the same element, the add wins leading to a state where the element belongs to the set. More formally, the set interface has two update operations: (i) $\mathsf{add}(e)$, for adding element e to the set; and (ii) $\mathsf{rmv}(e)$, for removing element e from the set. Given a set of update operations O that are related by the happens-before partial order $\prec$, the state of the set is defined as: $\{e \mid \mathsf{add}(e) \in O \wedge \nexists \mathsf{rmv}(e) \in O \cdot \mathsf{add}(e) \prec \mathsf{rmv}(e)\}$.

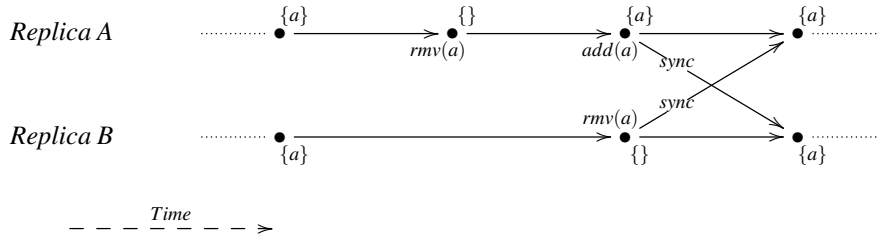


Figure 4.2.1: Run with an add-wins set.

Figure 4.2.1 shows a run where an add-wins set is replicated in two replicas, with initial state $\{a\}$. In this example, in replica A, a is first removed and later added again to the set. In replica B, a is removed from the set. After receiving the updates from the other replica, both replicas end up with element a in the set. The reason for this is that there is no $rmv(a)$ that happened after the $add(a)$ executed in replica A.

An alternative semantics based on the happens-before relation is the *remove-wins*. Intuitively, in the *remove-wins* semantics, in the presence of a concurrent add and remove of the same element, the remove wins leading to a state where the element is not in the set. More formally, given a set of update operations O, the state of the set is defined as: $\{e \mid \mathsf{add}(e) \in O \wedge \forall \mathsf{rmv}(e) \in O \cdot \mathsf{rmv}(e) \prec \mathsf{add}(e)\}$. In the previous example, after receiving the updates from the other replica, the state of both replicas would be the empty set, because there is no $add(a)$ that happened after the $rmv(a)$ in replica B.

Another relation that can be useful for defining the concurrency semantics is that of

a total order among updates and, particularly, a total order that approximates wall-clock time. In distributed systems, it is common to maintain nodes with their physical clocks loosely synchronized. When combining the clock time with a site identifier, we have unique timestamps that are totally ordered. Due to the clock skew among multiple nodes, although these timestamps approximate an ideal global physical time, they do not necessarily respect the happens-before relation. This can be achieved by combining physical and logical clocks, as shown by Hybrid Logical Clocks [58], or by only arbitrating a wall-clock total order for the events that are concurrent under causality [110].

This relation allows to define the *last-writer-wins* semantics, where the value written by the last writer wins over the values written previously, according to the defined total order. More formally, with the set O of operations now totally ordered by $<$, the state of a *last-writer-wins* set would be defined as: $\{e \mid \mathsf{add}(e) \in O \wedge \forall \mathsf{rmv}(e) \in O \cdot \mathsf{rmv}(e) < \mathsf{add}(e)\}$. Returning to our previous example, the state of the replicas after the synchronization would include a if, according the total order defined among the operations, the $rmv(a)$ of replica B is smaller than the $add(a)$ of replica A. Otherwise, the state would be the empty set.

(c) Key research findings

Preservation of sequential semantics When modelling an abstract data type that has an established semantics under sequential execution, CRDTs should preserve that semantics. For instance, CRDT sets should ensure that if the last operation in a sequence of operations to a set added a given element, then a query operation immediately after that one will show the element to be present on the set. Conversely, if the last operation removed an element, then a subsequent query should not show its presence.

Sequential execution can occur even in distributed settings if synchronization is frequent. An instance can be updated in replica A, merged into another replica B and updated there, and merged back into replica A before A tries to update it again. In this case we have a sequential execution, even though updates have been executed in different replicas.

Historically, not all CRDT designs have met this property. The *two-phase set* CRDT (2PSet), does not allow re-adding an element that was removed, and thus it breaks the common sequential semantics. Later CRDT set designs, such as *add-wins* and *remove-wins* sets, do preserve the original sequential semantics while providing different arbitration orders to concurrent operations.

Extended behaviour under concurrency Some CRDT designs handle concurrent operations by arbitrating a given sequential ordering to accommodate concurrent execution. For example, the state of a *last-writer-wins* set replica shown by its interface can be explained by a a sequential execution of the operations according to the LWW total order used. When operations commute, such as in G-Counters and PN-Counters, there might even be several sequential executions that explain a given state.

Not all CRDTs need or can be explained by sequential executions. The add-wins set is an example of a CRDT where there might be no sequential execution of operations to explain the state observed, as Figure 4.2.2 shows. In this example, the state of the set after all updates propagate to all replicas includes a and b, but in any sequential extension of

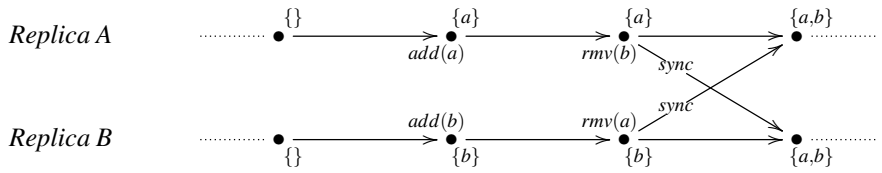


Figure 4.2.2: Add-wins set run showing that there might be no sequential execution of operations that explains CRDTs behavior.

the causal order a remove operation would always be the last operation, and consequently the removed element could not belong to the set.

Some other CRDTs can exhibit states that are only attained when concurrency does occur. An example is the *multi-value register*, a register that supports a simple write and read interface. If used sequentially, sequential semantics is preserved, and a read will show the outcome of the most recent write in the sequence. However if two or more values are written concurrently, the subsequent read will show all those values (as the *multi-value* name implies), and there is no sequential execution that can explain this result. We also note that a follow-up write can overwrite both a single value and multiple values.

Guaranties and limitations An important property of CRDTs is that an operation can always be accepted at any given replica and updates are propagated asynchronously to other replicas. In the CAP theorem framework [23, 54], the CRDT conflict-free approach favors availability over consistency when facing communication disruptions. This leads to resilience to network failure and disconnection, since no prior coordination with other replicas is necessary before accepting an operation. Furthermore, operations can be accepted with minimal user perceived latency since they only require local durability. By eschewing global coordination, replicas evolve independently and reads will not reflect operations accepted in remote replicas that have not yet been propagated to the local replica.

In the absence of global coordination, session guaranties [100] specify what the user applications can expect from their interaction with the system's interface. Both state based CRDTs, and operation based CRDTs when supported by reliable causal delivery, provide per-object causal consistency. Thus, in the context of a given replicated object, the traditional session guaranties are met. CRDT based systems that lack transactional support can enforce system-wide causal consistency, by integrating multiple objects in a single map/directory object [8]. Another alternative is to use mergeable transactions to read from a causally-consistent database snapshot and to provide write atomicity [83].

Some operations cannot be expressed in a conflict free framework and will require global agreement. As an example, in an auction system, bids can be collected under causal consistency, and a new bid will only have to increase the offer with respect to bids that are known to causally precede it. However, closing the auction and selecting a single winning bid will require global agreement. It is possible to design a system that integrates operations with different coordination requirements and only resorts to global agreement when necessary [64, 96].

Some global invariants, that are usually enforced with global coordination, can be

enforced in a conflict free manner by using escrow techniques [80] that split the available resources by the different replicas. For instance, the Bounded Counter CRDT [15] defines a counter that never goes negative, by assigning to each replica a number of allowed decrements under the condition that the sum of all allowed decrements do not exceed the value of the counter. While its assigned decrements are not exhausted, replicas can accept decrements without coordinating with other replicas. After a replica exhaust its allowed decrements, a new decrement will either fail or require synchronizing with some replica that still can decrement. This technique uses point to point coordination, and can be generalized to enforce other system wide invariants [14]

(d) CRDT models

There are two main CRDT models: state-based and operation-based (op-based). In op-based designs, the execution of an operation is done in two phases: prepare and effect. The former is performed only on the local replica and looks at the operation and current state to produce a message that aims to represent the operation, which is then shipped to all replicas. Once received, the representation of the operation is applied remotely using effect. On the other hand, in a state-based design an operation is only executed on the local replica state. A replica periodically propagates its local changes to other replicas through shipping its entire state. A received state is incorporated with the local state via a merge function that deterministically reconciles both states. To maintain convergence, merge is defined as a join: a least upper bound over a join-semilattice.

The two models are believed to be useful in both edge applications on heavy edge and light edge. Op-based CRDTs allow for simpler implementations, concise replica state, and smaller messages, but they assume a message dissemination layer that guarantees reliable exactly-once causal broadcast. This makes op-based CRDTs more suitable to heavy edge scenarios where the system model is less dynamic. On the other hand, state-based CRDTs are more robust to network changes and can be more suitable to light edge networks. The price is however in the communication overhead of shipping the entire state, which can get very large in size.

The following sections discuss optimizations for both models.

4.2.2 Towards operation-based CRDTs at the edge

In this section, we present our contributions to optimize the Pure op-based CRDT model that addresses the efficiency challenges in the classical op-based model. In particular, this model reduces redundant meta-data dissemination and storage. The main contribution to this model, that was introduced in [16], is presenting a portfolio of more than ten datatypes compare to three in the past. In addition, the new datatypes support the “reset” operation that required substantial modification to the original framework. Finally, we have provided new specifications and practical efficiency optimizations to the classical op-based CRDTs already implemented in Antidote DB.

(a) Pure op-based CRDTs

In op-based CRDTs, the execution of an operation is done in two phases: *prepare* and *effect*. The former is performed only on the local replica and looks at the operation and current state to produce a message that aims to represent the operation, which is then

shipped to all replicas. Once received, the representation of the operation is applied remotely using *effect*. Different replicas are guaranteed to converge as long as messages are disseminated through a reliable causal broadcast messaging middleware, and *effect* is designed to be commutative for concurrent operations.

In the standard approach, a *prepare* not only builds messages that duplicate the information already present in the middleware (even if it is not currently made available), but causality meta-data is often incorporated in the object state, hence, reusing design choices similar to those used in state-based approaches. Such designs impose larger state size and do not fully exploit causal delivery information. This freedom in current op-based designs is against the spirit of 'sending operations', and leads to confusion with the state-based approach. Indeed, in the standard op-based framework, a *prepare* can return the full state, and an *effect* can do a full state-merge (which mimics a state-based CRDT). This means that basically any CRDT could be described as op-based, even if it is for all purposes a state-based one, where no operation propagation is really happening. We believe that the above weakness and confusion can be avoided if the causality meta-data can be provided by the messaging middleware. Causal broadcast implementations already possess that information internally, but it is not exposed to clients. In the *pure operation-based CRDT* approach, initially introduced in [16], we propose and exploit such an extended API to achieve both simplicity and efficiency in defining op-based CRDTs.

In the *Pure* Op-Based CRDT framework, *prepare* cannot inspect the state, being limited to returning the operation (including potential parameters). The entire logic of executing the operation in each replica is delegated to *effect*, which is also made generic (i.e., not data type dependent). For pure op-based CRDTs, we propose that the object state is a *partially ordered log of operations – a POLog*. Causality information is provided by an extended messaging API: *Tagged Causal Stable Broadcast* (TCSB). We use this information to preserve convergence and also design compact and efficient CRDTs through a *semantically based POLog compaction* framework. The idea is to prune the POLog using a datatype-specific *obsolescence* relation, defined over timestamp-operation pairs. An example of obsolete operation is an *add* followed by another *add* of the same item in a set.

Following this line of research, we have recently obtained the following results that are summarized in the technical report [17], under submission to a journal paper:

- Improved the pure op-based framework, namely how obsolescence/redundancy is described, to support the *reset* operation in all types. This required revisiting the entire framework introduced previously in [16].

- Introduced a portfolio of several CRDTs (including variants of counters, sets, registers, maps, etc.) based on the pure op-based framework, and that can be used in several edge applications.

(b) Optimized classical op-based CRDTs

The pure op-based CRDTs, though advanced, they incur additional complexity compared to the classical ones. The tradeoff is however extra meta-data storage and dissemination as classical CRDTs often retain meta-data per operation forever. Antidote DB, a geo-replicated edge database that provides transactional causal consistency for CRDT [35]

(see deliverabl D6.1 or D6.2), opts for classical op-based CRDTs and only implements few of them. Our contribution was to improve and optimize these implementations as well as to implement new ones to enrich the Antidote DB CRDTs library.

In particular, for the existing data types, such as *PN-Counter*, *OR-Set* (*AW-Set*) and *RW-Set*, our improvements were in terms of minimizing the size of the downstream calculated to be broadcast to the other replicas and by that reduce the transmission overhead. In addition, we minimized the complexity of downstream messages/operations through general refactoring of the code. Finally, we also added some new op-based data types to Antidote's CRDTs library:

- Flags: We added an *EW-Flag* (Enable Wins Flag) and a *DW-Flag* (Disable Wins Flag) that store a Boolean value (*true* or *false*). Antidote has two variants for flags, which differ in how concurrent updates are resolved (i.e., which operation wins (*enable* or *disable*) in case those two were concurrent).

- Maps: Another data type that we implemented is a map called an *RR-Map*, which stands for Remove Resets Map, that offers a new design of a map data type than the already implemented *G-Map* (Grow only Map) and *AW-Map* (Add Wins Map). This map is implemented as an optimization on the size of downstream operations (both *update* and *remove*) as well as the size of the state stored locally at each replica when compared to the *AW-Map*. Both implementations were kept though as the semantics between the two designs differ a bit.

- *reset* operation feature: we implemented *reset* operation (where missing) in the data types because it is a nice feature and as well as it is required for data types that could be embedded in the *RR-Map*.

- Resettable counters: Implementing this *reset* feature was not possible for the existing op-based counter CRDT design as the effect of individual operations disappear—contrary to other datatypes in which operations retain tags. We implemented a naive resettable counter data type, i.e., *Fat-Counter*, as a map from individual operations to incremented value. To reduce the high storage overhead, we introduced an efficient version called Compact Resettable counter [107] which allows garbage collecting non-concurrent operations using the notion of *causal stability* explained earlier.

For all the data types, both existing and newly implemented, we added unit tests and property-based tests and implemented a feature of compressing operations before being disseminated to replicas.

4.2.3 State-based CRDTs at the edge

The alternative state-based CRDT model is also interesting for light edge computing where netowrks are hostile and dynamic. This is referred to the state (S) design that froms a join-semilattice where joining states is inflation: $s' \in S$ is an inflation of $s \in S$ if $s \sqsubseteq s'$; and therefore, the needed (partial order) meta-data is encapsulated in the datatype state regardless of the middleware. Consequently, states can be merged in an ad-hoc way and converge despite duplications, e.g., due to retransmission likely to occur in such networks. In this section, we convey our work on synchronization optimization of delta

CRDTs: an efficient variant of state-based CRDTs introduced previously in [7]. Our contribution was to reduce the redundant dissemination as well as the overhead of state transfer when nodes (replicas) are intermittently synchronizing or new replicas join the edge newtwork. We first overview delta CRDTs.

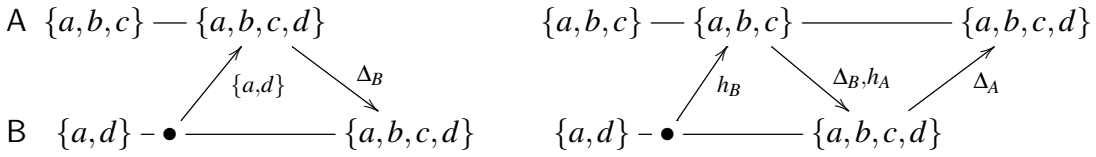
(a) Delta CRDTs

In state-based CRDTs, the entire state is periodically propogated and merged with other replicas. As the local state grows, this approach becomes impractical. Delta state-based CRDTs only propagate the most recent modifications incurred in its local state. This is achieved by building delta-mutators that are variants of mutators (update operations) that return smaller state, called delta δ, representing the recent state change. When the delta is merged with the local state, the same inflation is produced as if the original mutator was applied. In this model, the δs resulting from mutators are added to a δ-buffer, in order to be propagated to neighbor replicas in the next synchronization step. When a δ-group (a batch of deltas) is received from a neighbor, it is also added to the buffer for further propagation.

(b) Efficient Synchronization of State-based CRDTs

Problem. The above buffering and dissemination scheme can incur redundant propoagation of deltas either through back propagation (sending a delta to its source) or transitive propagation (sending deltas as part of a delta-group to those who have already seen them). Consequently, each replica to keep track of which δ-group in the δ-buffer has been effectively received by its neighbors. When a δ-group is acknowledged by all neighbors, it is removed from the buffer. If a neighbor stops acknowledging (e.g., due to a network partition), the buffer will grow indefinitely, which might force garbage-collection of its content. Hence, the metadata required for delta-based synchronization is not available (this also occurs in systems with dynamic networks, where the set of neighbors is constantly changing). A naive solution proposed in current delta-based synchronization [7] is to perform bidirectional full state transmission. This brings us back to square one, i.e., to the state-based model.

Proposed solution. We introduced two methods to avoid bidirectional full state transmission: state-driven and digest-driven. Taking a Grow-only set (GSet) example, Figure 4.2.3 sketches the synchronization algorithms. In state-driven approach (Figure 4.2.3a), B starts by sending its local state to A, and given this state, A is able to compute a Δ_B that reflects the updates missed by B. Convergence is achieved after two messages (if no updates where performed in the meantime), since A can merge the state received from B into its local state. In the *digest-driven* approach (Figure 4.2.3b), instead of sending its local state, B starts by sending to A a digest of its local state h_B (smaller than the local state), that still allows A to compute Δ_B. Upon the receipt of h_B, replica A sends the computed Δ_B and a digest h_A of its local state, so that B can also compute Δ_A. As a consequence of starting the synchronization with a digest, three messages are necessary to achieve convergence.



(a) *State-driven* synchronization (b) *Digest-driven* synchronization

Figure 4.2.3: Synchronization of a *grow-only set* with two replicas A, B.

Join Decomposition. For the success of the above synchronization methods, an efficient technique is required to compute the difference between two states in an optimal way (to avoid redundancy). To this end, we introduced *join-decomposition* that can be informaly defined as follows: a set s' that belongs to the power set of a join semilattice state s is a join-decomposition iff the join of all elements in s' is necessary to produce s and all elements are join-irreducible (cannot be decomposed further). Said differently, it is the set of primitive undecomposable elements of a set that are necessary and sufficient to build a state. As an example, for a semillatice $s = \{a,b,c\}$, $\{\{a\},\{b\},\{c\}\}$ is a join-decomposition, whereas $\{\{b\},\{c\}\}$ and $\{\{a,b\},\{c\}\}$ are not.

The concept of join-decomposition allow us to improve the synchronization of state-based CRDTs in two distinct situations: when a replica has the necessary metadata for δ-based synchronization, we *remove redundant state in received δ-groups*; when this metadata is not available, replicas can employ *state-driven* or *digest-driven* synchronization, avoiding bidirectional full state transmission. A detailed description of this work can be found in MSc thesis [44].

4.3 Communication support for data at the Edge

The data management techniques and datatypes discussed in the previous two sections assume the presence of underlying dissemination layer with properties that support general cloud and edge applications (e.g., causality), as well as networks (e.g, scalability and dynamicity). In particular, the work on advanced Pure op-based CRDTs assume the presence of a causal middleware that is efficient and supports "causal stability" which is novel to SOTA causal middlewares [17, 18, 88]. We started developing a middleware that supports these features as we present next. On the other hand, all the dissemination protocols used in the previous sections are implemented using Distributed Erlang that has known to limits to parallelism and no support for cluster topologies (but *full mesh*). We developed a communication library called Partisan in a previous project (FP7 Syncfree [34]) that implements two group membership protocols, i.e., Plumtree [86] and HyParView [63], that are efficient hybrid gossip protocols. In LightKone, we developed Partisan further to support edge networks and edge applications. In particular, we supported dynamic network topologies, many application patterns, multiple channel sending, etc. Even on a lower layer, are working with Ericsson to develop the Distributed Erlang library to support mesh networks with routing instead of relying on full connected mesh doomed unscalable (the number of connections in the cluster grows N2 with the number of nodes). We present the progress of the mentioned contributions in the following, and

we mention the steps to be addressed in future milestones.

4.3.1 Tagged Causal Stable Broadcast (TCSB)

Problem. As stated in the previous section, the pure op-based framework relies on two important features in the underlying dissemination layer. The first is causal delivery in which sent messages must only be delivered in respect with the causality relation (a.k.a., "happened-before" relation [61]) between those messages. Although state of the art causal delivery protocols [18, 88] (also known as group multicast protocols) provide this functionality, additional meta-data are needed (in the CRDTs) to express the concurrency between messages which is not provided by the aforementioned protocols. This redundant meta-data leads to an overhead on the dissemination of messages and therefore on the performance and scalability of the system. The second feature is "causal stability" used to discard timestamp information of operations: a timestamp t, and corresponding message, is causally stable at node i when all messages subsequently delivered at i will have timestamps $t' > t$. Stability can be locally detected by tracking in each node the last timestamps received from each other node. The notion is novel and has been introduced in [16], but no details has been mentioned about how to implement this feature in a causal middleware.

Solution. To address the above issues, we developed novel causal delivery middleware called Tagged Causal Stable Broadcast (TCSB). A common implementation strategy for a reliable causal broadcast service is to assign a vector clock to each message broadcast, and use the causality information in the vector clock to decide at each destination when a message can be delivered. However, there is no ordering provided by RCB for concurrent messages, even though the middleware can detect when messages are concurrent. Our solution of solving this problem with no duplication of effort and meta-data, is by exposing this knowledge itself to the application. When the application is provided by the causality relation between operations, correct semantics are preserved with less effort and redundancy.

Moreover, TCSB provides a "causal stability" oracle that informs on stable messages, those with no further concurrent deliveries. The causal stability information is stored in a matrix-like data structure of size *nxn* and the stable threshold calculated from this matrix reduced to a vector of size n, n being the number of participants in the group. When this threshold of causal stability (we call it stable vector) is calculated, it means everything below this vector is stable. This mechanism allowed garbage collection and compaction of partially-ordered logs.

We have an implementation of the TCSB, written in Erlang that we mention in the Software chapter 5.

Feasibility and future plan. TCSB can be used for any datatype model that requires reliable causal delivery. The API is standard; it supports the usual broadcast and delivery APIs in addition to novel ones that could be exploited in cases where application-level meta-data needs the internal causality information of the middleware (e.g., timestamps) or even do garbage collection. Therefore, TCSB can be used mainly at heavy edge to support classical op-based CRDTs as well as efficient ones as pure op-based CRDTs.

Through our work on TCSB, we found that exposing the causal relation between messages to the application could seem very trivial at first glance. However, this "exposing" of the causality meta-data is tricky, and could lead to many pitfalls if not done with care. We plan in the future to discuss those pitfalls to help and guide implementors be aware of them and to provide a simple, generic and elegant design and implementation of Tagged Causal Delivery and Stability that provides the necessary meta-data to applications (such as CRDTs) for correct semantics and behavior. In addition, We aim to provide an improved version of this TCSB where causality is based on *dots* instead of vector clocks. (A dot is a lightweight meta-data representing a pair of actor id and a local counter.) We expect that this new implementation would be more efficient than the vector clock based one and helps us scale better to fit edge scenarios.

4.3.2 Partisan

Problem. Despite the pervasiveness of distributed applications, runtime support for building cloud and edge/fog tailored distributed applications remains rare, requiring application developers to build and maintain a communications framework in addition to their application code. While not yet the norm in industry, there are some notable counter examples, all of which are implementations of a distributed actor model; for example: Akka Cluster [2], Microsoft Orleans [25], and Distributed Erlang [106]. Each of these frameworks enables transparent distributed programming for the platforms they are designed for, but all three optimize for a single type of application: low-latency, small-object messaging between nodes in a single cluster, operating inside the data center, using the *full mesh* model— which is not scalable ($O(N^2)$) with the number of nodes in the network N. Therefore, an edge network composed of two or more network topologies may be desired without paying this price in Distributed Erlang (in which the majority of artifacts in LightKone are implemented). Thus, there is a need for a communication library that supports multiple communication patterns and network topologies (e.g., server/client, peer-to-peer, Publish/Subscribe, etc.) at once.

Solution. We developed Partisan, an alternative distribution layer and distributed programming model for Erlang and Elixir. Partisan is designed to be used instead of, or alongside of, Distributed Erlang, and supports multiple cluster topologies, all of which can be specified at runtime. Partisan supports five different topologies that are common in edge networks: full mesh (with or without Distributed Erlang), client-server, peer-to-peer, static, and publish-subscribe. To be able to support all of these different topologies in a single system, Partisan does not attempt to provide feature parity with Distributed Erlang, but instead, presents a smaller programming API that can be supported by all of the different topologies efficiently. Partisan provides a rich API for users with standard operations for viewing cluster membership, joining and removing nodes from the cluster, and asynchronously delivering messages to other nodes in the cluster.

Partisan also addresses the problem of the "one-size-fits-all" topology: since, Distributed Erlang assumes one network topology for cluster communication, the design cannot be optimal for all types of edge and distributed applications. Therefore, Partisan allows the user to specialize these topologies at runtime, choosing the most appropriate topology for the application at hand. The support topologies are as follows:

- *Static.* In static mode, Partisan will only connect to other nodes that have been explicitly configured at the time of node deployment time.

- *Full Mesh.* In full mesh mode, Partisan will ensure all nodes in the cluster are fully connected; in that, each node will connect to every other node in the cluster directly, ensuring each node has full knowledge of the entire cluster. This topology is an implementation of the default configuration of Distributed Erlang.

- *Client-Server.* In client-server mode, Partisan will ensure that all nodes tagged as clients only connect to nodes tagged as server; and all nodes tagged as server nodes will connect to one another. Client-server is an implementation of the traditional topology used by rich-web and mobile applications.

- *Peer-to-Peer.* In peer-to-peer mode, Partisan will have all clients connect to one other client in the system and the resulting network will approximate an Erdös-Rényi model.

- *Publish-Subscribe.* In publish-subscribe mode, Partisan will connect to preconfigured AMQP message broker for node-to-node messaging and dissemination of cluster membership information.

Partisan work is under submission. The code is open-source freely available on GitHub (see Chapter 5), and has several industry adopters.

Feasibility and future plan. Partisan implements, as a backend, HyParView [63] and Plumtree [86] hybrid gossip membership and overlay protocols that are known of their efficiency in large peer-to-peer (as light edge networks). They use spanning trees to disseminate payload and flooding to exchange small mete-data to reconstruct the tree when failures occur. As applications written using Partisan can modify the topology at runtime, Partisan can be used as a basis for trying out various types of cluster topologies, and running comparisons between them, as we explore and identify the most appropriate topologies for the edge computing environment. Partisan is currently used as backend to Lasp [73] and overlay layer for TCSB mentioned earlier. In the future, we plan to evaluate Partisan empirically to measure its scalability to the number of edge nodes.

4.3.3 Erlang Communication Support for Edge computing

From its origin as a Telecom language, the Erlang Virtual machine (EVM) has been designed from the ground-up with the ideas of concurrency, availability and distribution. As such it offers built-in tools for designing scalable systems, as it has been demonstrated during the last decades. Unfortunately, edge computing requires the EVM to be able to support clusters with a very large number of nodes, and the current implementation shortcomings prevent such a use-case.

In details, the EVM is limited in the number of nodes that can take part of a cluster, because it is designed as a full-mesh graph of interconnected nodes. Consequently, the number of connections in the cluster grows N^2 with the number of nodes. A solution to support the scale of distribution required by the edge computing use-case is to add to the EVM a mesh distribution model where messages would be routed through multiple nodes

to their destination. Because in a mesh model the messages are routed through multiple nodes and the topology can change over time, we lose the strict message delivery ordering guarantee required by the EVM. In addition, the well-known EVM issue with head-of-line message blocking gets even worse when a message needs to pass through multiple nodes to reach its destination. We discuss these problems and the corresponding solutions we developed in the two following sections.

(a) Message Strict Ordering

Problem. An Erlang process requires the guarantee that all the messages it receives from another process are received in the order they have been sent. This guarantee is easily fulfilled in the current implementation, because all nodes are interconnected and all the message between two nodes are serialized in a single TCP connection. In the case of a mesh network, the changes in topology could result in messages reaching their destination in a different order they have been sent. Any mesh distribution model should be able to detect out-of-order messages and re-order them before delivering them to the destination processes. Even though the current implementation provides node-level message ordering, we only need process-level message ordering. Having a finer-grained ordering requirement prevents a lost or delayed message between two processes to block all the messages between other processes routed through the same node.

Solution. We developed a proof-of-concept distribution model for the EVM available on Github (see the Software chapter 5). The current proof-of-concept achieves solves the above issue by grouping messages in channels, a channel being defined by a pair of process identifiers in the case of normal inter-process messages. All the messages for a given channel have a strictly monotonically increasing sequence number that is used by the destination node to enforce the message ordering. In addition of the inter-process messages, the EVM needs to support messages where the recipient is not a process. These messages are used to provide services like process and node monitoring, and connection health checks (tick messages). The distribution model prototype handles these messages by using different channels when possible and falling back to a special out-of-band channel when not. In the particular case of the tick messages used to monitor the health of the connections, they are sent with the highest priority, short-cutting the message reordering layer. The implications of losing global node-level message ordering guarantee is still under investigation, but the preliminary results from the proof-of-concept distribution model show that it should not break the usual Erlang assumptions.

(b) Head-Of-Line Blocking

Problem. A well-known issue with the current EVM distribution model implementation, that sending large messages blocks all the other concurrent messages to the same destination node. This is called the head-of-line blocking issue. In a mesh network, where the messages travel through multiple nodes, the delay introduced by this issue accumulates along the path.

Solution. In order to have a predictable latency for message delivery, our solution introduced in the proof-of-concept distribution model is to fragment all the messages in small

chunks and schedule them fairly so no large message blocks the other ones. Given the distribution model prototype already has the concept of channel for messages, it schedules the messages per channel. The result is that processes sending very large messages will never block other processes sending small ones, even if the destination is the same process. At this stage, the fragments are sent following a round-robin scheduling per channel.

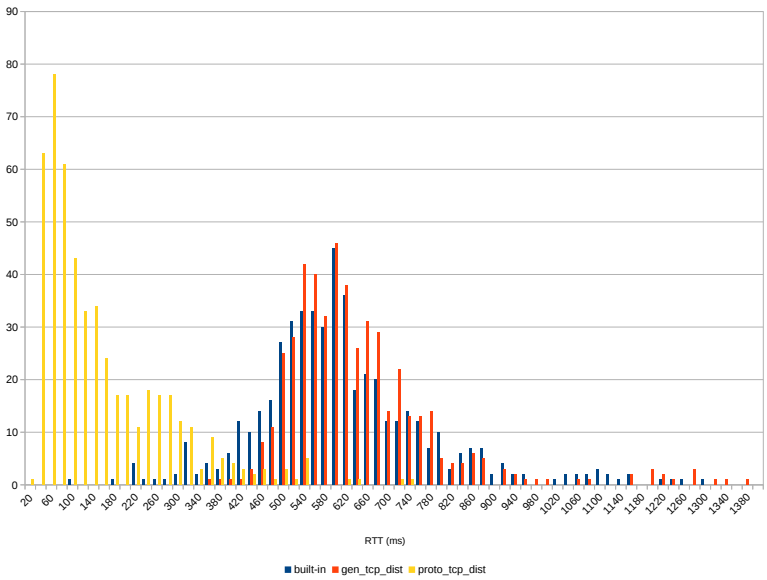


Figure 4.3.1: RTT distribution of 500 samples

Additional pedantic detail. The proof-of-concept distribution model we developed takes advantage of a new EVM API currently only available as a patch for OTP 20.0.4 that should be officially released with OTP 21. With this API it is possible to develop an EVM distribution layer entirely in Erlang without the need for low-level driver.

We have also experimented these features through a Head-of-Line Blocking benchmark. The benchmarking environment consists of two EVM nodes connected through Wi-Fi using either the built-in distribution, gen_tcp_dist distribution module or the prototype implementing message fragmenting proto_tcp_dist. gen_tcp_dist is an example provided with the API patch that replicate the exact behavior of the built-in distribution using the new API, it is used to compare the eventual cost of using an Erlang-level distribution module. The benchmarking setup is to start 10 process pairs exchanging a binary message of a given size between the two nodes. The benchmark in itself consists of measuring the average time for a synchronous call between the two nodes.

The preliminary results of the benchmarking of the current proof-of-concept distribution model shows that packet fragmenting greatly reduces the impact of the head-of-line blocking issue when large messages are being sent by other processes. Figure 4.3.1 shows the distribution of the RTT for the different distribution backends.

(c) Feasibility and future plan.

The above features are important contributions for the efficiency of Distributed Erlang. Being at the EVM level, these contributions are generic and could be helpful to any distributed edge computing protocol using Erlang, among those used in LightKone. Currently, the prototype focuses on resolving the head-of-line blocking issue. Even though it provides support for out-of-order messages, it is not yet required because there is no mesh networking and message routing at this point. In the future, we aim at supporting UDP-based distribution protocols. We are currently working together with Ericsson to improve the Erlang custom distribution API in Erlang/OTP. The current OTP API is very much connection oriented, which means we need to fake connections to it even if we are using UDP. What we hope to learn by implementing distribution in UDP is how the API needs to be extended to make a connectionless transport work smoothly.

4.4 Scalable Data Management at the Edge

Data scalabilty is an essential feature to support edge networks and applications. There are at least two scalability dimensions of particular interest at the edge: storage and network size. The former differs from classical cloud systems given the relateively limited storage capacities of edge devices, even those at the heavy edge like micro data centers [55, 68, 95]. A natural technique to address this challenges is to use partional replication (a.k.a., partitioning or sharding). SOTA data partioning is inadequate to edge systems due to the overhead of metadata and voluminous payloads disseminated especially when some properties, like causal consistency, are needed [66, 67, 111]. In this vein, we introduce two solutions to address data partitioning through reducing the metadata disseminated, as explained next in Saturn, or even avoid sending the payload if unnecessary, as explained next in nonuniform partial replication. The other dimension is addressing the increazing network size likely in edge networks and applications. In particular, highly available datatypes as CRDTs can only scale to few tens of nodes due to the incured metadata overhead [7, 17, 84]. To that end, we introduce two techniques that provide highly scalable counters using hierarchical containment (i.e., Handoff Counters) or transient identity borrowing (i.e., Borrow Counters). These techniques, as detailed in the following sections, are inspiring approaches to develop more sophisticated datatypes beyond counters in the next part of the project.

4.4.1 Saturn

Problem. The problem of ensuring consistency in applications that manage replicated data is one of the main challenges of distributed computing. The observation that delegating consistency management entirely to the programmer makes the application code error prone and that strong consistency conflicts with availability has spurred the quest for meaningful consistency models, that can be supported effectively by the data service. Among the several invariants that may be enforced, ensuring that updates are applied and made visible respecting causality has emerged as a key ingredient among the many consistency criteria and client session guarantees that have been proposed and implemented in the last decade. Mechanisms to preserve causality can be found in systems that offer from weaker to stronger consistency guarantees. In fact, causal consistency is pivotal in

the consistency spectrum, given that it has been proved to be the strongest consistency model that does not compromise availability. To maintain causality, data is often associated with metadata, e.g., timsetamps or token tags. Previous solutions either favor throughput by compressing metadata into a single scalar [42], penalizing remote visibility latency; or favor remote visibility latency by using more precise ways of tracking causality: using more metadata that usually is not constant but dependant on the number of objects [66, 67], or the number of datacenters [3, 111].

Solution. Saturn [22] is a novel metadata service that can be used by geo-replicated data services to efficiently ensure causal consistency across geo-locations. Its design brings two main contributions:

- It eliminates the tradeoff between throughput and data freshness inherent to previous solutions. To avoid impairing throughput, our service keeps the size of the metadata small and constant—namely labels, independently of the number of clients, servers, partitions, and locations. By using clever metadata propagation techniques, we also ensure that the visibility latency of updates approximates that of weak-consistent systems that are not required to maintain metadata or to causally order operations.

- It allows data services to fully benefit from partial geo-replication, by implementing genuine partial replication, requiring datacenters to manage only the data and the metadata concerning data items replicated locally.

Saturn is solely responsible of propagating labels (the metadata associated to each update operation) among datacenters, delivering them in an order that respects causality. The service assumes that the update payload is propagated among locations by means of some bulk-data transfer scheme that fits the application business requirements. A datacenter makes remote updates visibile to local clients in the order in which the labels are received. For this, it has to have received not only update's corresponding label, but also the update's payload (tagged with the label).

Saturn exploits the fact that causal order is a partial order to deliver different serialization of labels to each datacenter with the goal of minimizing remote update visibility latency: the time that takes for updates to become visible in remote datacenters. To achieve this, Saturn relies on a tree-based dissemination architecture. This architecture is composed by a set of *serializers* organized in a way that the metadata paths (latency between datacenters when traversing the tree) matches (when possible) the data paths (latency among datacenters through the bulk-data transfer scheme). In addition, labels include information with regard to the data being updated. Based on this information, Saturn can selectively deliver labels to only the set of interested datacenters, enabling genuine partial replication.

In our experiments, Saturn has demonstrated to add negligible overhead in terms of throughput and remote visibility latency when attached to data services that enforce no consistency in both full and partial replicated settings with a handful set of datacenters.

Feasibility and future plan. Nevertheless, Saturn was designed for datacenter-based could computing. Therefore, its design expects a handful set of powerful, stable replicas. Instead, in edge computing, we expect a potentially large set of heterogeneous, unstable

resources. Thus, Saturn would require multiple modifications to work on the edge computing paradigm. Saturn needs to find the optimal architecture given the set of replicas, such that the latencies among replicas through the tree and through the data paths differ minimally. This is at the moment done by means of a constraint solver that finds the optimal configuration among all possible. This is expensive and not solvable in polinomial time. When increasing the number of replicas, we will have to find a suboptimal mechanism that it is capable of scaling up to a large number of replicas. In addition, Saturn does not consider replicas failing or simply being offline constantly. Thus, its default reconfiguration mechanism requires recomputing the configuration considering all replicas again (as we expect this to happen very infrequently). This may be to expensive if failures occur frequently. We believe a method to repair the tree in a reactive way is required such that only small changes to the tree are applied. The same applies to new replicas joining. Finally, Saturn permits clients to migrate among replicas. This is crucial when having partial replication: clients may need to read data that it is not replicated in their local replica. In Saturn, consistent migration is implemented in a simple but not so efficient manner. This is because in Saturn, we assume that the replicas are datacenters with capability for storing most of the data their local clients may require, which makes the migration operation infrequent. Nevertheless, under edge computing, the replicas placed on the edge of the network will have significantly less resources than a datacenter. This will force clients to migrate more often in order to read data that it is not replicated locally. Thus, Saturn's migration mechanism needs to be optimise to offer a good quality-of-service to clients when operating on the edge.

4.4.2 Nonuniform replication

Problem. When adopting partial replication, each replica only maintains part of the data. As a consequence, each replica can only locally process a subset of the database queries. In geo-replicated scenarios, where each data center only maintains part of the data, for executing a query it might be necessary to contact one or more remote data centers, leading to high latency for executing operations. In edge scenarios, where edge replica necessarily maintain only a subset of the data, this problems becomes more important.

Solution. We propose an alternative partial replication model, the non-uniform replication model, where each replica maintains only part of the data but can process all queries. The key insight is that for some data objects, not all data is necessary for providing the result of read operations. For example, an object that keeps the top-K elements only needs to maintain those top-K elements in every replica. However, the remaining elements are necessary as one of the elements not in the top needs to be promoted when a top element is removed. This top-K object could be used for maintaining the leaderboard in an online game.

In our work, we have: formalized the concept of non-uniform replication; applied the model to replicated systems that provide eventual consistency; derived sufficient conditions for providing non-uniform eventual consistency; and defined CRDTs that adopt the non-uniform replication model.

For formalizing the non-uniform replication model, we started by defining that two object states, s_i and s_j, are *observable equivalent*, $s_i \overset{o}{\equiv} s_j$, iff the result of executing every

read-only operation on both states is equal, i.e., $\forall o \in \mathcal{Q}, o(s_i) = o(s_j)$, with $\mathcal{Q}$ the set of read-only operations. A replicated system is non-uniform if the replication protocol guarantees that in a quiescent state, i.e., after the propagation of all relevant updates to all replicas, the state of any two replicas is observable equivalent.

The non-uniform replication model can be used in replicated systems that provide eventual consistency. We say that a replicated system provides *non-uniform eventual consistency* iff in a quiescent state the state of any replica is observable equivalent to the state obtained by executing some serialization of O, with O the set of operations executed. As a consequence, the state of any pair of replicas is also observable equivalent.

For a given set of operations in an execution O, we say that $O_{core} \subseteq O$ is a set of core operations of O iff $s^0 \bullet O \stackrel{o}{\equiv} s^0 \bullet O_{core}$, with s^0 the initial state of all replica, i.e., the states obtained by executing a serialization of operations of O and O_{core} in the same initial state is observable equivalent.

A replication system provides *non-uniform eventual consistency (NuEC)* if, for a given set of operations O, the following conditions hold: (i) every replica executes a set of core operations of O; and (ii) all operations commute.

Given these sufficient conditions, we have derived an algorithm for defining which operations need to be propagated to other replicas. The algorithm locally classifies operations in four groups: the operations that will never have impact in the observable state; the operations that have impact in the observable state, as computed locally; the operations that might have impact in the observable state, depending on the operation that have been executed in other replicas; the operations that might have impact in the observable state in the future. Only the operations in the second and third group need to be propagated, as proved by Cabrita et. al. [26].

Finally, we have applied this concept to operation-based CRDTs, by developing useful operation-based CRDTs that adopt this model (NuCRDT).

Feasibility and future plan. The goal of non-uniform replication is to allow replicas to store less data and use less bandwidth for replica synchronization. Although it is clear that non-uniform replication cannot be useful for all data, we believe that the number of use cases is large enough for making non-uniform replication interesting in practice. We now discuss two classes of data types that can benefit from the adoption of non-uniform replication.

The first class is that of data types for which the result of queries include only a subset of the data in the object. In this case two different situations may occur: (i) it is possible to compute locally, without additional information, if some operation is relevant (and needs to be propagated to all replicas); (ii) it is necessary to have additional information to be able to decide if some operation is relevant.

An example of the former is a Top-K CRDT that allows access to the top-K elements added to the object. This object can be used, for example, for maintaining the leaderboard in online games or for maintaining materialized views of partitioned data. Another example includes a data type that returns a subset of the elements added based on a (modifiable) user-defined filter – e.g. in a set of books, the filter could select the books of a given genre, language, etc.

An example of the latter is the Top-Sum NuCRDT that maintains the top-K elements added to the object, where the value of each element is the sum of the values added for the element. This data type can be used for maintaining a leaderboard in an online game

where every time a player completes some challenge it is awarded some number of points, with the current score of the player being the sum of all points awarded. It could also be used for maintaining a top of the best selling products in an (online) store (or the top customers, etc). Another example includes a data type that returns the 50^{th} percentile (or others) for the elements added – in this case, it is only necessary to replicate the elements in a range close to the 50^{th} percentile and replicate statistics of the elements smaller and larger than the range of replicated elements.

In all these examples, the effects of an operation that in a given moment do not influence the result of the available queries may become relevant after other operations are executed — in the Top-K with removes due to a remove of an element in the top; in the filtered set due to a change in the filter; in the Top-Sum due to a new add that makes an element relevant; and in the percentile due to the insertion of elements that make the 50^{th} percentile change. We note that if the relevance of an operation cannot change over time, the non-uniform CRDT would be similar to an optimized CRDT that discard operations that are not relevant before propagating them to other replicas.

A second class is that of data types with queries that return the result of an aggregation over the data added to the object. An example of this second class is the Histogram CRDT presented in the appendix. This data type only needs to keep a count for each element. A possible use of this data type would be for maintaining the summary of classifications given by users in an online shop. Similar approaches could be implemented for data types that return the result of other aggregation functions that can be incrementally computed [79].

A data type that supports, besides adding some information, an operation for removing that information would be more complex to implement. For example, in an Histogram CRDT that supports removing a previously added element, it would be necessary that concurrently removing the same element would not result in an incorrect aggregation result. Implementing such CRDT would require detecting and fixing these cases.

As discussed in deliverables of WP5 and WP6, we have integrated the non-uniform replication model in AntidoteDB and are studying how to adopt it in the frameworks developed specifically to run in the light-edge.

4.4.3 Handoff counters

Problem. A problem that can easily arise in state-based CRDTs is scalability: if there is one replica per participating entity, each one with a unique identity, as many CRDTs keep maps with these ids as keys, these maps will keep growing over time, as more entities participate; this results in each replica having size proportional to the total number of entities that ever participated in the system, which is not scalable. This will be specially problematic in edge-computing use cases that include large numbers of entities.

One possible approach to address scalability is to restrict replicas to a small number of server nodes, excluding clients from the participating entities, in what concerns the CRDT. In the case of edge computing, this would mean choosing some nodes as "servers", making all the others be clients. This will solve the scalability problem and allow unreliable communication between server nodes, but will not solve the fault-tolerance problem in the client-server interaction. This is because a basic problem with counters is that the increment operation is not idempotent; therefore, an increment request by a client (which itself does not keep a replica) cannot just be re-sent to the server in case

there is no acknowledgment. This approach would be poor in terms of fault-tolerance, not exploiting system-wise the good fault-tolerance properties of state-based CRDTs.

Solution. Handoff Counter is a state-based counter that we have developed, which addresses both scalability, availability and fault-tolerance, allowing each participating node to be a replica. The mechanism is described in detail in a journal paper [5], accepted for Springer Distributed Computing. In brief, the main ingredients of our approach are:

- a replication mechanism that is non-symmetric (contrary to standard CRDTs where all replicas converge to the same state);
- hierarchical tiered topology that allows availability through alternative communication paths and different roles, e.g., many pure transient clients, a few permanent persistent servers in datacenters, and many intermediate-tier nodes spread over the edge;
- node identity containment. Contrary to typical CRDTs, node ids are not propagated to the whole network, being only temporarily stored in the state of some neighbor nodes;
- a mechanism to perform a reliable handoff of a value between two nodes, possibly using a third party, to achieve tolerance to partitions or transient node failures.

The mechanism allows arbitrary numbers of nodes to participate and adopts the CRDT approach, having a replica at each node without distinguishing clients and servers (therefore, overcoming the problems of server-side CRDTs), and allowing an operation (fetch or increment) to be issued at any node. It addresses the scalability issues (namely the id explosion in version-vectors used in traditional counters) by: assigning a tier number (a non negative integer) to each node, in a hierarchical structure, where only a small number of nodes are classified as tier 0; having "permanent" version vector entries only in (and for) tier 0 nodes, therefore, with a small number of entries; having a *handoff* mechanism which allows a tier $n+1$ "client" to handoff values to some tier n "server" (or to any lower tier node, in general); making the entries corresponding to "client" ids be garbage-collected when the handoff is complete.

Additional pedantic details. The essence of the handoff mechanism relies on two components of the replica state, one containing what we call *slots* – each slot serves as a capability of receiving a value – and the other contains *tokens* – a token holds a value and matches a single slot. The general idea is that, over time: a slot is created in the state of a node j (destination of the handoff); a token matching that slot is created in another node i (source of the handoff) to which some value (number of increments) accounted locally is moved; the slot at j is "filled": the slot is removed and the value in the corresponding token is acquired by j (added to the locally accounted value); node i removes the token. The mechanism ensures correctness no matter what communication patterns may occur. Towards this, each node also keeps a pair of counters: the *source clock* and the *destination clock*. Figure 4.4.1 presents a run where a node i hands off some value (9 in this example) to a node j, when no messages are lost. The non-zero values in the

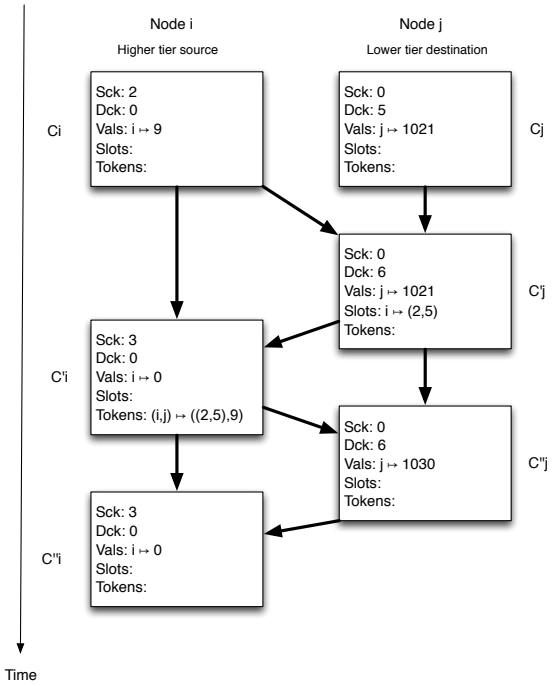


Figure 4.4.1: A handoff from node i to j (only relevant fields are shown).

source clock of node i and destination clock in node j indicate that the run is a continuation of a longer run that already did some handoffs from i to j. Several properties are ensured, namely:

- A given slot cannot be created more than once; even if it was created, later removed and later a duplicate message arrives;
- A token is created specifically for a given slot, and does not match any other slot;
- A given token cannot be created more than once; even if it was created, later removed and later a duplicate message having the corresponding slot arrives.

We have both proved the mechanism correct, and performed experimental evaluation, which confirmed the good scalability and fault-tolerance properties. Two implementation of the mechanism where made, one in Clojure, publicly available at https://github.com/pssalmeida/clj-crdt, and another in Rust, available at https://github.com/pssalmeida/handoff_counter-rs. To efficiently evaluate scenarios with up to one hundred thousand clients, we built a discrete event simulator for asynchronous networks using the Rust programming language. The simulation infrastructure, including the scripts used to obtain the results (as well as the runs performed) can be obtained from https://github.com/pssalmeida/handoff_counter_simulator-rs.

Feasibility. As we discuss in the journal paper, the underlying mechanism and lessons learned are applicable far beyond simple counters. We have devised a mechanism which allows some value to be handed off reliably over unreliable networks, through multiple paths to allow availability in the face of temporary node failures or network partitions. Values are moved from one place to another by "zeroing" the origin and later "adding"

to the destination. Reporting is made by aggregating in two dimensions: "adding" values and taking the "maximum" of values. The value accounted at each node is updated by a commutative and associative operation which "inflates" the value. Given the above, the handoff counter CRDT can be generalized to any commutative monoid; and one can inspire from it to design more useful and complex datatypes in the future.

4.4.4 Borrow Counters

Another way to address edge network scalability is to exploit the behaviour of nodes in dynamic settings to circumvent the identity explosion problem. Instead of the generic hierarchical design of Handoff Counters, here we propose a simple two-layered design, distinguishing only *permanent* nodes (e.g., datacenter nodes) and *transient* nodes (e.g., end-clients). The overall idea is that, when a transient node joins the system, it asks to borrow an identity from a permanent node. While in the system, it uses this borrowed identity to increment the counter. If the transient node wishes to leave the system, it marks the borrowed identity as an inactive. When a permanent node observes inactive identities, it performs garbage-collection of these entries.

The Borrow Counter [46] makes use of the Causal CRDT [7] concept to achieve the transfer of increments from transient to persistent nodes in an elegant way, allowing node retirement without incurring a permanent impact on state growth. Figure 4.4.2 shows an example with two nodes a and b: node a acts as permanent and b as transient. Node a starts by creating dot a_1 for itself, and later on dot a_2 for node b; node a increments the counter by 9, and b by 8; node b disables its dot and node a transfers node b increments to its entry in the Borrow Counter. (Here we are denoting inactive dots by bold numbers, and representing the causal context in the Borrow Counter by its maximal entries, i.e. $\{a_1, a_2, c_1\} \equiv \{a \mapsto 2, c \mapsto 1\}$).

$$
\begin{array}{rc}
s_0 = \bot & (\{\},\{\}) \\
 & \Big| \\
s_1 = \mathsf{create}_a(s_0, a) & (\{a \mapsto \{a_1 \mapsto 0\}\}, \{a \mapsto 1\}) \\
 & \Big| \\
s_2 = \mathsf{create}_a(s_1, b) & \begin{array}{l}(\{a \mapsto \{a_1 \mapsto 0\}, \\ \quad b \mapsto \{a_2 \mapsto 0\}\}, \{a \mapsto 2\})\end{array} \\
s_3 = \mathsf{inc}_a(s_2, 9) & \begin{array}{l}(\{a \mapsto \{a_1 \mapsto 9\}, \\ \quad b \mapsto \{a_2 \mapsto 8\}\}, \{a \mapsto 2\})\end{array} \\
s_4 = \mathsf{inc}_b(s_3, 8) & \\
s_5 = \mathsf{retire}_b(s_4) & \begin{array}{l}(\{a \mapsto \{a_1 \mapsto 9\}, \\ \quad b \mapsto \{a_2 \mapsto \mathbf{8}\}\}, \{a \mapsto 2\})\end{array} \\
 & \Big| \\
s_6 = \mathsf{transfer}_a(s_5, b) & (\{a \mapsto \{a_1 \mapsto 17\}\}, \{a \mapsto 2\})
\end{array}
$$

Figure 4.4.2: Borrow Counter example with two nodes: node a is permanent, while node b acts as transient

Additional pedantic details. A Borrow Counter is a Causal CRDT where the *dot store* is a DotMap [7] from node identifiers to another dot store. This other dot store is a

DotFun [7] mapping dots to the lattice pair $\mathbb{B} \times \mathbb{N}$ (where $\mathbb{B}$ is the set of booleans, and $\mathbb{N}$ the set of naturals). A node can increment the counter if its entry in the map has at least one active dot (dots created by permanent nodes) *i.e.* at least a dot in the DotFun mapped to a $(\mathsf{False}, \_)$ pair. When a transient node retires, it marks all its dots as inactive. This transition from active to inactive is irreversible, as given by the $\mathsf{False} < \mathsf{True}$ lattice used in the pair. Once a transient node makes a dot inactive, it surrenders the capability of issuing further increments to that entry, allowing a safe subsequent transfer to the permanent node that created it.

Feasibility. Borrow Counters are an effective technique to address the issue of identity explosion of large edge networks. In particular, it is very convenient to situations that incur transient nodes as in dynamic networks (e.g., mobile networks or Vanets). In addition, the idea of Borrow Counters can be generalized to design other datatypes in the same manner. In our experience, most CRDT datatypes can be designed as such. We plan to investigate this topic further in the remaining part of the project.

Chapter 5

Software Deliverables

In this section, we present the software deliverables, libraries, and components in which the contributions presented in this report appear. Most of these software are direct artefacts that show in LiRA or used as backend components and libraries. Since this work package is meant to provide the support to build generic edge computing runtimes, we believe that developing fine-grained components is crucial to increase the impact of LightKone's work on external edge computing platforms. Indeed, although LiRA perfectly fits the set of LightKone use-cases, the latter represents a sample edge computing set of applications, and thus addressing more use-cases may require building other edge computing runtimes. To this end, the components provided in this deliverable can be used in building new edge runtimes or integrated in existing ones to leverage the technology LightKone provides.

- AntidoteDB: is a geo-replicated CRDT data store offering transactional causal consistency and fits nicely the heavy edge. It is publicly available under Apache 2.0 license. Available at https://github.com/SyncFree/antidote.
- Lasp: a framework that allows to design and execute scalable synchronization-free applications that resort to CRDTs as their data model. Available at https://github.com/lasp-lang/lasp.
- Antidote CRDT library: an Erlang library of operation-based CRDTs comprising counters, flags, maps, sets, integer, registers and sequence (RGA). The library can be used in any system, and is currently used in Antidote. Available at https://github.com/SyncFree/antidote_crdt.
- Legion: a framework that allows to develop web-based client applications that leverage of edge computing interactions among clients running the same application in a transparent way. Available in https://github.com/albertlinde/Legion.
- Partisan: a TCP-based membership system written in Erlang/Elixir and implements the HyParView[63], hybrid partial view membership protocol, with TCP-based failure detection. Partisan is suitable for dynamic topologies being robust and lightweight. Available at https://github.com/lasp-lang/partisan.
- Efficient CRDTs library: a library of state-based, delta-based, delta-composition, and pure-op-based CRDTs written in Erlang. The library includes implementations

of grow-only counter, positive-negative counter, lexicographic counter, bounded-counter, grow-only set, two-phase set, add-wins and remove-wins set, enable-wins and disable-wins flag, last-writer-wins and multi-value registers, and an add-wins map. Available at https://github.com/lasp-lang/types.

- TCSB middleware: a vector-based middleware that implements the TCSB (Tagged Causal Stable Broadcast) protocol, written in Erlang. The implementation supports efficient causal delivery and causal stability. Available at https://github.com/gyounes/trcb_base.
- proto_dist: is an Erlang-level distribution protocol using the new EVM API in Erlang 21 to fix the head-of-line blocking issue with standard distribution. Available at https://github.com/Stritzinger/proto_dist, but access is provided upon request.

Chapter 6

Security Analysis of Use-cases

Among the prime motivations behind edge computing is the potential to provide better security and privacy guarantees. Although important, LightKone's contribution on security is limited to analyzing the security threats and requirements of LightKone use-cases (presented in deliverable D2.1). The corresponding solutions can then be using standard off-the-shelf security methods and tools or through developing novel solutions. Since LightkKone's focus is mainly on data managements and communication, we will only highlight the open issues and potential threats, recommend off-the-shelf security measures, and develop new solutions to only part of these threats—as defending against DoS attacks in WP5. A use-case-specific analysis will be presented in the future deliverable to cover the entire software stack.

6.1 Overview

From the use cases description and security analysis in D2.1, it is possible to assume that from the security point of view, the most significant challenges and open issues are located into the *Light-edge* scope. In this way, the majority of use-cases (UCs) share requirements that involve balancing data integrity, confidentiality, availability and authentication with some kind of constraint in such entities. These constraints are generically related to device capabilities or undefined surface with external entities (*i.e.*, the boundaries between the system and its environment).

Considering this panorama, some research topics might benefit an encompassing number of the proposed UCs. An early list is summarized below and further discussed along security observations for each UC:

- *Lightweight cryptography for Light-edge environments* aiming to provide confidentiality in data flows among constrained entities;
- *Homomorphic encryption algorithms* for operations over encrypted data, such as aggregation and queries at the edge;
- *Cross-platform software sandboxing* in order to ensure contained operations in third-party and heterogeneous entities;
- *DoS identification and prevention* at communication level for heterogeneous architectures.

Following subsections discuss specific security aspects in each UC.

6.2 UPC - Guifi.net community network

Although security issues are not a main concern, according to UC description in D2.1 report, some aspects might be pointed out. The most significant is the lack of a detailed Threat and Adversary Model, including their number, definition, acceptable frequency and severity. Some conflicting requirements also seem to impact security properties in such system. In this way, some challenges and suggestions encompassing the three use cases are firstly presented below.

From the system architecture, it is relevant to understand how entities interact with each other, for instance, *(i)* is there any authentication mechanism between the central database and the monitoring servers or between these servers and those monitored nodes? *(ii)* how do the monitoring servers collect data from the nodes (*e.g.*, SNMP)?; *(iii)* How often do entities communicate?

Considering monitoring servers as the main source of threats, some mechanisms might be deployed, for instance, *(i)* a more controlled process of instantiating a new *SNPService*, following predefined security policies and *(ii)* a more sophisticated access control, such as those based on public key certificates, which will improve authenticity, integrity and accountability attributes.

From each UPC's user case security analysis in D2.1, some challenges are listed in following subsections.

6.2.1 Coordination between servers & Data storage for the monitoring system

- Considering that *SNPService* is running at the premises of the users, avoiding data tampering requires software verification and mechanisms able to prevent users from changing/producing data collected from nodes;

- Task assignment might be performed based on consensus strategies, although some threats regarding serve collusion will still remain;

- Assigning two monitoring servers to each network node is not enough in order to guarantee integrity of collected data. Collusion attacks or a Byzantine server are examples in which this approach fails. In addition, monitoring systems based in pooling protocols, such as SNMP, will provide divergent measurement data due to its asynchronous nature. This aspect will impact directly the cross-checking suggested process;

- Replicating the local measurements with other monitoring servers increases the system resilience and availability, however it raises some different threats, such as data tampering or omission. As measurement data is only consumed by the central database, it might be encrypted with the DB public key before being transmitted to other serves. A major challenge here is to comply with the aggregation process described in Section 2.5, as well as the computational burden involved in public key cryptography in *light-edge* nodes (*e.g.*, Single-Board-Computers described in

Chapter 3). For the first challenge, an interesting research direction is resorting to homomorphic encryption [78][98], which might stress the second challenge;

- Although the security analysis state that data protection is not a concern, users consumption patterns (monitored for billing purpose) must be protected before replicating across servers.

6.2.2 Service provision support for the Cloudy platform

- It is necessary to introduce some mechanism preventing users from writing service data objects;
- A small challenge in this UC involves preventing service provides from manipulating published service data objects for their own benefit. Currently everyone is allowed to manipulate the entries;
- Considering that Guifi.net administrators can not arbitrate provided services (as in *SNPService*), preventing underlying service providers from manipulating billing information seems to be a major challenge in this UC.

6.3 Scality

6.3.1 Pre-indexing at the edge

The UC provides a reasonably clear threat model (a severity and tolerance analysis could improve it), however the assumption that software and hardware platform at the client premises are secure does not seem realistic, unless the client application provides some kind containment mechanism. Some of the described threats might be exploited through third-party application or at the communication level, mainly non-encrypted data and pre-indexes.

Based on UC's description, the objectives at security level consist in providing integrity and confidentiality guarantees to data, metadata and query responses at different levels across the system. However, from the LightKone point of view, an interesting showcase might be the metadata availability in case of network partition or divergent data provoked by some malicious entity into the system tampering data (even unintentionally).

For queries on encrypted data, an interesting research direction might involve homomorphic encryption, although the underlying computational burden might be an obstacle.

6.3.2 Lambda functions at the edge

This UC is quite similar to the previous one, with the addition of a new asset to be protected, *i.e.*, Lambda functions, which might be running either in *light-edge* or *heavy-edge* devices. Assuming that Lambda functions are running in isolated sandbox environments, ensuring the correctness of each function (its integrity) and that they are properly authenticated in order to access only the resources required in such computation seems to be the major challenges, beyond those discussed in previous description.

6.3.3 S3 local cache of central data

Although being architecturally similar to previous UC, this one has a less clear *Threat and Adversary Model*, for instance, the privileges and access control of each entity caching data or manipulating those already cached. Even so, the UC description provides a more clear idea on which are the challenges identified by the partner, *i.e.*, ensure cached data integrity and prevent any unauthorized entity from eavesdropping on or tampering local data.

Considering possible research challenges, a local mechanism able to prevent central synchronization of unauthorized or maliciously deleted/tampered data seems to be an interesting topic.

6.4 Stritzinger

6.4.1 No-Stop RFID

Due to resource constraints of RFID readers, the communication between RFID tags, readers/writers (cache) is unencrypted and solely relies on efficiency of firewall rules at the higher level of the network and physical access control at the factory. This model might be improved based on lightweight cryptographic algorithms and certificates for confidentiality, authentication and key distribution in order to prevent, for instance, an employee or external supplier attacking the system with cheaper RFID writers. Lightweight cryptography represents a growing research topic with interesting recent works from industry and academia [43, 47, 72].

This direction might tackle with some reported threats, such as, *(i)* protecting cache data from eavesdropping or manipulation by unauthorized parties; *(ii)* ensuring that only legitimate tags provide data to the system; *(iii)* verifying the authenticity of RFID reader software and hardware.

Considering the WP 5 scope, the objectives related to security and resilience to DoS attacks in communications between RFID readers seem to be a valuable showcase for LightKone project.

6.4.2 Smart Metering Gateway

Although it seems to be a promising overall showcase for LightKone project, from the security perspective, most of threats are out of Stritzinger's scope, mainly due to the intrinsic dependence of WM-Bus standard and what metering sensor suppliers provide in this field. Even the communication between gateways relies on such approach. The same situation is observed when considering data privacy, a highly sensitive aspect into this UC.

Even so, some interesting topics can be pointed out, namely *(i)* authenticity verification of software updates for meters and gateways might resort to selective reprogramming (*e.g.*, Deluge, MNP and MOAP); *(ii)* integrity, authentication and accountability might resort to certificates (public or private); *(iii)* Aggregating encrypted data aiming to reduce traffic overhead also might resort to homomorphic encryption; *(iv)* protecting gateways from DoS attacks involves tailored IPS/firewalling rules.

6.4.3 Swarm of Small Satellites

There is no security analysis for this use case in D 2.1.

6.5 Gluk - Agriculture Sensing Analytics

Globally, the most challenge aspect regarding the security analysis in this UC is the balance between all flexibility required in Chapter 6, Section 1.3 with security requirements in Section 9.5. For instance, the usage of general purpose of-the-shelf sensors (*e.g.*, Arduino/Raspberry Pi based), with effortless deployment, relying solely on the security features provided by such hardware seem incompatible with both the listed security requirements and the described thrust model.

In this way, a more detailed threat and adversary model should be provided, including their probabilities, severity and acceptable levels. A better description of the assets to be protected, such as sensors, communication channels, security mechanisms in use, protocols and services are also indispensable for a clear understanding on this field.

Regardless the aspects aforementioned, some generic challenges can be identified:

- use of lightweight authentication and cryptography mechanisms in *light-edge*;
- provide secure communication channels among diverse device architectures (at communication level);
- increase availability through energy aware processes. This aspect is already being addressed into LightKone scope, *i.e.*, LiteSense scheme;
- define and provide a secure domain in such flexible environment;
- preventing DoS at the communication layer.

Chapter 7

Advancing State of the Art

In this chapter, we provide a literature review of the works related to LightKone. We cover reference architectures, distributed data management, and communication protocols. Throughput the text, we show the advances of LightKone in this work package for Year 1. A summary of the contributions is also presented in Table 7.0.1 for the convenience of the reader. Considering the overall plan presented in Chapter 4 and the effort conducted on LiRA, we estimate the current progress regarding the final milestone to be between 30% and 40%.

Table 7.0.1: Summary of LightKone WP3 Contribution for Year 1.

Component	Description	Previous SOTA	Contribution	Software	Reference
State CRDT	State-based data management for relaxed consistency at the edge	Not generic enough; few datatypes	Generic framework; many datatypes; join-decomposition	Legion, Lasp	Cha 4. Sec. 4.2
Op CRDT	Operation-based data management for relaxed consistency at the edge	Non-edge-tailored datatypes; few datatypes; not generic	Generic framework; optimized edge-tailored datatypes; support resets; many datatypes; resettable counters	AntidoteDB	Cha 4. Sec. 4.2
Scalable Counters (Handoff and Borrow)	Datatypes scalable with the number of edge nodes or dynamicity	blocking sync datatypes; ID explosion	scalable counters using hierarchical trees; transient IDs for counters; single writer principle	None	Cha 4. Sec. 4.4
Saturn	Partial replication (sharding) meta-data handling with causality support	Causal multicast protocols; Cure protocol for causal delivery	Reduced metadata propagation for enforcing causality; timely delivery of updates	None	Cha 4. Sec. 4.4
Causal Delivery and stability middleware	middleware for causal consistent systems and op-based CRDTs	Redundant meta-data in causal delivery	Reduced meta-data in causal delivery; causal stability concept	TCSB	Cha 4. Sec. 4.3
Distributed Communication	Edge-tailored alternatives for distribution layer for Erlang.	Plumtree, HyParView; Erlang distribution	Partisan: Hybrid gossip-based with different net topologies and various clusters; mesh-based EVM.	Lasp, proto_dist	Cha 4. Sec. 4.3
Computation CRDTs	CRDTs for which the state is the result of a computation over the executed operations (e.g. aggregation results), adopting the non-uniform replication model	Distributed aggregation protocols	Non-uniform replication model; integrates computations with the storage	Antidote-DB	Cha 4. Sec. 4.4

7.1 LightKone Reference Architecture (LiRA)

State of the art of LightKone reference architecture (LiRA) is addressed in details in Chapter 3. For completeness, we summarize the relation to state of the art focusing on the data management aspect that is a key contribution of in LightKone. Being a promising extension to cloud computing, fog and edge computing have been very active areas in research and industry. Consequently, several edge/fog architectures have been proposed in SOTA like Open Fog [33], Edge Foundry [51], Microsoft Azure IoT [75], Amazon IoT Green grass [11], ETC Edge Computing [32], and ETSI MEC [48]. However, there is an existing gap in all these architectures in the data management level of the application layer in which data cannot be efficiently shared/replicated unless through an upper layer intermediary (a higher layer fog node or the cloud center). This represents a single point of failure and imposes unacceptable response time to edge applications. The main innovation in LiRA is the support for generic application-level data and computation through developing artifacts and software components that support replicated data that is highly available and proven to converge at once. Importantly, LiRA allows data sharing across the hierarchy of the edge system as well as at the same layer. In addition, LiRA artifacts span a wide spectrum of heavy, medium, and light edge/fog devices thus supporting a wide range of applications and patterns.

LiRA is compatible and complementary to SOTA architectures. For instance, the OpenFog RA (ORA) is a generic architecture that set standards and recommendations to the required features and properties at the entire software stack. It emphasizes the importance of autonomy and availability without proposing solutions to them at the application layer as in LiRA. In the discussed use cases, ORA highlights the difficulty of data sharing at the same edge layer, which LiRA provides in particular. On the other hand, Microsoft Azure IoT is based on a time streaming where data is basically pushed to the cloud center for processing. To improve response times, a ”warm” database is used to provide data for edge IoT devices mainly, for a recent date and time range, aggregated data for one or many devices, etc. Therefore, there is no support for generic edge applications semantics or data management at the edge/fog layer as we do in LiRA. ECC Edge Computing follows the Azure IoT approach ensuring a fast time series (centralized) database that stores immutable data associated with timestamp for speed.

Contrary to Azure IoT and ECC Edge Computing, Amazon IoT Greengrass extends Amazon’s AWS cloud to edge devices, allowing edge devices to run AWS Lambda functions, execute predictions based on machine learning models with or without connection to the cloud center. Edge nodes can also integrate with third party applications but without data sharing as in LiRA. EdgeX Foundry is another edge framework maintained by Linux Foundation with the ambition to be a key edge/fog open source platform for IoT applications. Data in EdgeX is only handled across layer (north-west) leaving unilateral data management to a future plan. Other platforms like FIWARE FogFlow and ETSI MEC do not make use of replicated data and thus focus more on data at the cloud, databases, or corresponding dissemination.

7.2 CRDTs

Conflict-free Replicated Data Types (CRDTs) are data abstraction tools that can be particularly helpful in edge computing scenarios due to allowing for high availability and autonomy by design. State of the art CRDT has been however developed mainly in Syncfree [34] project to address the geo-replicated scenario. The designs are however costly and incomplete on the edge. In particular, the op-based CRDTs used in AntidoteDB [35] were not edge-tailored and required optimizations to improve their efficiency. The same holds for Pure op-based CRDTs [16] and state-based CRDTs [84] in which meta-data has to be reduced to the fit the edge model. Furthermore, SOTA CRDTs [7, 16, 84] can only scale to few tens of replicas which is not convenient to edge network. We tried to address develop Handoff Counter datatype [5] with hierarchical containment that can scale to hundreds of nodes and preserve idempotence. We plan to support more datatypes inspiring from this approach. We addressed this challenge through Borrow Counters [46] that allow transient nodes to join and leave the system without losing their data. This model is planned to be extended to all datatypes in the future deliverables. Finally, CRDTs currently assume a fixed number of replicas which not the case of dynamic edge networks. This will also be addressed in LightKone in this WP.

7.3 Communication support

The data management techniques and datatypes discussed in the previous two sections assume the presence of underlying dissemination layer with properties that support general cloud and edge applications (e.g., causality), as well as networks (e.g, scalability and dynamicity). Despite the myriad of work in the area of causal broadcast, anti-entropy, and distributed protocols in Erlang, most of these work do not satisfy the needs of the data structures we developed in LightKone. We show how we advance beyond state of the art in these areas.

7.3.1 Causal Multicast

The work on Pure op-based CRDTs assume the presence of a causal middleware that is efficient and supports “causal stability” which is novel to SOTA causal middlewares.

Several mechanisms exist for implementing reliable causal multicast. Those protocols may be implemented inside a group communication system (GCS) (e.g., Isis [19, 20], Transis [12], JGroup [74], Spread [13], JGroups [56],...). The first protocols, initiated by the CBCAST [20] protocol from Birman and Joseph [20], included in each multicast message its own causal history; i.e., a set of precedent messages not yet delivered in all system processes. This led to many Subsequent designs such as Psync [81] and works by Ladin et al. in their lazy replication proposal [59]. That was simplified when vector clocks [50, 71] were introduced, reducing the size of the multicast messages. Some amount of causal information (the vector clocks themselves) is still kept, in proportion of the system size. The first protocols developed for the vector-clock-based approach by Birman et al. in [19] (anticipated by Schiper et al. [89] and Raynal et al. [85], though for point-to-point communication) turned out to be efficient and scaled better than those

of the previous generation. Other subsequent proposals are based on this kind of causal history information. For instance, Schiper and Pedone [91] propose a protocol for open groups. Almeida et al. [4] propose a mechanism for bounding the size of the elements used in version vectors. Also, Almeida et al. solution [6]: interval tree clocks (ITC), which deals with system membership changes. The main problem of this new generation of protocols was the length of the vector clocks. A first solution was already presented by Birman et al. [19] and Stephenson [99]: it consisted in compacting clocks by recording only incremental changes. Inspired in the compacting approach from [19], a more general compacting solution was proposed by Singhal and Kshemkalyani [94]. It was later refined and formalized by Prakash et al. [82] and Kshemkalyani and Singhal [57]. The degree of compaction of these solutions was evaluated by Chandra et al. [29]. Another compacting solution had been described by Mostéfaoui and Raynal [77] in 1993.

Our work on TCSB, complements the existing reliable causal multicast/broadcast (RCB) systems: there is no ordering provided by RCB for concurrent messages, even though the middleware can detect when messages are concurrent. We use this fact, to allow the application, without any duplication of effort, to detect concurrency between operations which is needed for correct semantics and behaviour. Moreover, our work implements causal stability, a mechanism different than traditional stability notions, which also allows state compaction and garbage collection.

7.3.2 Erlang distributed protocols

Despite the pervasiveness of distributed applications, runtime support for building cloud and edge/fog tailored distributed applications remains rare, requiring application developers to build and maintain a communications framework in addition to their application code. While not yet the norm in industry, there are some notable counter examples, all of which are implementations of a distributed actor model; for example: Akka Cluster [2], Microsoft Orleans [25], and Distributed Erlang [106]. Each of these frameworks enables transparent distributed programming for the platforms they are designed for, but all three optimize for a single type of application: low-latency, small-object messaging between nodes in a single cluster, operating inside the data center, using the *full mesh* model—which is not scalable ($O(N^2)$) with the number of nodes in the network N. Chechina et al. [31] identified two challenges that must be overcome in Distributed Erlang to scale to hundreds of nodes. Specifically, (i) transitive connection sharing, and (ii) explicit process placement.

While these changes enable Scalable Distributed Erlang to break through the scalability bottleneck with global operations previously identified by Ghaffari et al. [52, 53], scaling up to 256 nodes [30] , these solutions still assume that explicit process naming through the global registry is desirable, from an application developer point of view. Additionally, a node that participates in too many groups also will fall into the same trap of replicating too much information. In LightKone, we developed a communication library called Partisan in a previous project (FP7 Syncfree [34]) that implements two group membership protocols, i.e., Plumtree [86] and HyParView [63], that are efficient hybrid gossip protocols. In LightKone, we developed Partisan further to support edge networks and edge applications. In particular, we supported dynamic network topologies, many application patterns, multiple channel sending, etc. On a lower layer, we are working with Ericsson to develop the Distributed Erlang library to support mesh networks with

routing instead of relying on full connected mesh doomed unscalable (the number of connections in the cluster grows N2 with the number of nodes).

7.3.3 Anti-entropy

In the context of anti-entropy gossip protocols, *Scuttlebutt* [103] proposes a *push-pull* algorithm to be used to synchronize a set of values between participants, but considers each value as opaque, and does not try to represent recent changes to these values as deltas. Other solutions try to minimize the communication overhead of anti-entropy gossip-based protocols by exploiting either hash functions [40] or a combination of Bloom filters, Merkle trees, and Patricia tries [24]. Still, these solutions require a significant number of message exchanges to identify the source of divergence between the state of two processes. Additionally, these solutions might incur significant processing overhead due to the need of computing hash functions and manipulating complex data structures, such as Merkle trees.

Our work on state synchronization is inspired by *rsync* [102] that synchronizes two files placed on different machines, by generating file block signatures, and using these signatures to identify the missing blocks on the backup file. In this strategy, there's a trade-off between the size of the blocks to be signed, the number of signatures to be sent, and size of the blocks to be received: bigger blocks to be signed implies fewer signatures to be sent, but the blocks received (deltas) can be bigger than necessary. Inspired by *rsync*, *Xdelta* [69] computes a difference between two files, taking advantage of the fact that both files are present. Consequently the cost of sending signatures can be ignored and the produced deltas are optimized.

7.4 Partial and Non-uniform Replication

Full replication protocols as in [9, 41, 66, 67, 70, 87, 105] can improve the reliability and availability of systems through making redundant data available across all replicas and thus allowing applications to share data. Updates are however costly and complex as described above. Partial replication [10, 37, 90, 101] addresses the issues of high update latency in full replication by having each replica store only part of the data. However, ensuring applications invariants remains a challenge. Among the several invariants that may be enforced, ensuring that updates are applied and made visible respecting causality has emerged as a key ingredient among the many consistency criteria and client session guarantees that have been proposed and implemented in the last decade.

Previous solutions either favor throughput by compressing metadata into a single scalar [42], penalizing remote visibility latency; or favor remote visibility latency by using more precise ways of tracking causality: using more metadata that usually is not constant but dependent on the number of objects [66, 67], or the number of datacenters [3, 111]. Saturn [22] is a metadata service we developed to efficiently ensure causal consistency across geo-locations and keeps the size of the metadata small and constant. Saturn allows data services to fully benefit from partial replication, by implementing genuine partial replication, requiring heavy edge nodes to manage only the data and the metadata concerning data items replicated locally. When adopting partial replication, each replica only maintains part of the data. As a consequence, each replica can only

locally process a subset of the database queries.

In a replicated scenarios, where a server only maintains part of the data, for executing a query it might be necessary to contact one or more remote replica, leading to high latency for executing operations. In edge scenarios, where edge replica necessarily maintain only a subset of the data, this problems becomes more important. In this report, nonuniform replication is proposed as an alternative partial replication model where each replica maintains only part of the data but can process all queries. The key insight is that for some data objects, not all data is necessary for providing the result of read operations. On the other hand, state of the art replication protocols cannot solve this problems as they either enforce strong consistency [36, 70, 76], weak consistency [9, 39, 66, 67, 105] or a mix of these consistency models [65, 97]. However, our work combines non-uniform replication with eventual consistency to improve the availability of the system, thanks to CRDTs. In our work, we formalized the concept of non-uniform replication; applied the model to replicated systems that provide eventual consistency; derived sufficient conditions for providing non-uniform eventual consistency; and defined CRDTs that adopt the non-uniform replication model.

Chapter 8

Exploratory Research

In this chapter, we present the research exploratory work that is related but not at the core of LightKone. These works have the potential of more exploration or inspiration in the future. For instance, the Single-Writer principle discusses the cases where some CRDTs may not incur concurrency. On the other hand, we include two security works for Privacy-aware IoT Data Management at the light edge and Byzantine resilient protocol for heavy and possibly light edge.

8.1 The Single-Writer Principle in CRDT Composition

Obtaining more powerful CRDTs by composition of more basic ones has been addressed in various ways, one example being the *causal CRDTs* [7] such as maps, in which the values are themselves CRDTs. In such generic compositions, each individual component must be designed in a way that it may be updated by multiple participants. However, there are usage scenarios where each component is semantically tied to a given participant, which is the sole updater. One example is Doodle[1], an online scheduling tool for meetings, used by a group of people to decide on a date. Each participant selects a set of desirable dates and Doodle aggregates the responses from all participants, finding the dates that will work best for everyone. In this scenario, each participant changes its own value in the scheduling page, adding or removing dates. Using current general CRDT designs (e.g., a map from participants to sets) is unnecessarily complex, as it does not exploit this single-writer scenario, where conflicting operations over each component will never occur. The *single-writer principle* in concurrent programming, where each register is only written by a single process, is a powerful concept, leading to simplifications or elegant designs, as the classic Bakery algorithm [60].

In [45] we show how the single-writer principle can be used to construct a new class of CRDTs in which conflicting updates never occur by design. We also define a generic collection of single-writer versioned objects along with two concrete collections. While this construction is classic in many systems and can be already found in the design of some flavors of CRDT counters, it was hidden inside specific implementations. Making this pattern an explicit composition construct, that can be used to make CRDTs from any sequential data-type, provides a new tool for safe construction of complex CRDT compositions.

[1] https://doodle.com

8.2 Security for the Edge

8.2.1 Privacy-aware IoT Data Management

IoT devices nowadays employ a "cloud-first" approach with all the collected sensor data being sent directly to the cloud of service provider for further processing and storage. Despite the benefits such cloud-based approach provides, the continuous data collection and retention in the cloud can have potentially negative consequences for the end users. First of all, the sensor data might be too sensitive to be processed remotely without the user awareness and control. For instance, IP camera video feed can expose the personal life of the users to unauthorized parties, and even simple electricity meter readings can reveal users' timetable and the home appliances they own. Secondly, service providers can share sensitive user data with third parties or use these data for targeted advertisement without the user's consent. Finally, the end users often have little or no knowledge of what data are actually being collected by the devices they own and how these data can later be used. They are thus forced to blindly trust the service provider with their sensitive information, which raises concerns and fears over data privacy.

To address the rising privacy concerns of the IoT users, an edge computing model was recently proposed [21], which aims to bring data processing closer to the user and the edge of the network. Such approach allows to minimize the volume of IoT-generated data flowing to the data center, reduce the network load by aggregating the data at the edge, and improve the user experience and application performance through a lower latency and response time. But most importantly, such approach allows for a better data privacy, since raw sensor data are processed "in-house" under the control of the end user. The edge node therefore has to be designed in a way to provide the data privacy guarantees and offer a flexible platform for service logic execution.

To address the consequences of ever growing number of IoT devices and the amount of traffic they generate, an edge computing model was recently proposed [21], which aims to bring data processing closer to the edge of the network. In this model, edge nodes with sufficient storage and computational capabilities are placed closer to the deployed IoT devices. These nodes perform local data processing and storage and send the results of the computation to the cloud of service provider for further processing and aggregation if needed. Such approach allows to minimize the volume of IoT-generated data flowing to the data center, reduce the network load by aggregating the data at the edge, and improve the user experience and application performance through a lower latency and response time. But most importantly, such approach allows for a better data privacy, since raw sensor data are processed "in-house" under the control of the end user. The edge node therefore has to be designed in a way to provide the data privacy guarantees and offer a flexible platform for service logic execution.

Over the last few years, multiple designs of IoT edge nodes, or hubs, have been proposed. Some of these offer a way to control how the sensor data is collected, processed, and shared by service providers, and allow to block operations that can result in privacy breaches [38, 49, 108]. This is done either by inspecting the data flow from IoT sensors to the data center in the cloud (variation of taint tracking or context analysis), or by providing a secure sandbox environment and limiting the scope of data operations. While these systems offer a better control over the sensitive user data, they often suffer from overtainting issues that lead to false-positives or even false-negatives allowing malicious

data flows to exist unnoticed. Furthermore, a limited set of data processing operations might not be flexible enough for various IoT scenarios.

Considering the limitations of the existing solutions, we propose HomePad [109] - a privacy-aware IoT hub that allows for a fine-grained control over the sensitive IoT data. Similarly to commercially available IoT hubs it provides a platform for execution of third-party apps that can interact with IoT devices and offer a certain service. The difference comes from the way these apps are written and executed. HomePad apps use a HomePad API to access and process the sensor data, and run in a confined environment which prevents any sensitive data leakage.

In HomePad, applications are implemented as a *directed graph of elements*. An element is an instance of a function that runs in isolation from other elements. Such an element-based structure allows to represent any application as a graph of elements it consists of. Such an approach forces application developers to make both all internal data flows and data transformations within their applications explicit. As a result, HomePad contributes to making applications more transparent with respect to how they access and process user data. Since every API element performs a certain function with the predefined input and output data types, it is possible to trace the flow of specific sensor data within the application's graph.

Furthermore, HomePad goes beyond existing IoT hub solutions by allowing the user to check automatically whether a given application satisfies his or her privacy requirements at install time. Graph-based application representation allows to analyze the sensor data flow within the app from a certain source (e.g. a frame from an IP camera) to a certain sink (e.g. module that issues HTTP network requests).

The installation of applications found to be violating the user privacy will be halted and the information about that will be added to the final report provided to the user. This report will contain an information about the sensitive sensor data an application has access to, the operations it performs and, most importantly, whether the application violates any of the privacy policies. The main goal of the application verification process is to make users aware of how their sensor data is accessed and processed, and eventually prevent the installation of applications that the user may deem to be too privacy-invasive.

Considering that some IoT scenarios require access to substantial computation and storage resources that are often not available at the edge, we further extended the HomePad's functionality to the private cloud of the user or a trusted service provider. For that, HomePad relies on Intel Software Guard Extensions (SGX) [1] secure enclave to ensure confidentiality and integrity of the sensor data.

We implemented a prototype of HomePad and used it to build several use case applications. From our evaluation of the system, we found that HomePad was able to effectively detect illegitimate data flows and incurs low performance overhead.

8.2.2 As Secure as Possible Eventual consistency

Available/Partition-tolerant systems favor availability over strong consistency and thus exhibit one or more variants of (relaxed) eventual consistency (EC) models. However, in literature, such systems often consider the fault-recovery model and thus cannot handle arbitrary or malicious faults. In this work, we introduced a new protocol [93] that integrates eventual consistent systems with a Byzantine Fault Tolerance cluster [27]. The system is designed to respect the essence of eventual consistency: availability is always

favored over security—which is done in the background—unless the client chooses otherwise. The protocol currently targets the heavy edge network whereas future work will investigate its feasibility on light edge.

The context Replicated services that are built through EC are highly available since client's requests are served via a local application server (or replica) without immediate synchronization with other servers; this step is however performed in the background to avoid blocking of client requests, but still ensure (eventual) data convergence. State-of-the-art research in EC assumes that replicas can crash and recover back to the last "healthy" state. Unfortunately, there is evidence that malicious and arbitrary (a.k.a., Byzantine [62]) faults are not rare even in leading Internet services [104]. In the case of EC, a Byzantine server can apply operations in an incorrect way (deliberately or not) which hampers data convergence, and thus compromises the entire service. Consequently, secure EC solutions that are resilient to Byzantine faults, being the strongest fault model [27], are highly advocated when the deployment conditions of servers and clients creates risk for this class of faults.

Why current solutions fall short? Classical BFT state-machine replication protocols protocols [27] cannot simply solve the EC problem due to two main reasons. The first is that such protocols are often blocking to the clients since total order coordination is required per operation. The second reason is that replicas are considered *correct* (i.e., not Byzantine) as long as all replies match; i.e., it requires that replies are exactly equivalent. In a recent work [28], the authors tried to solve the latter case by allowing a replica to immediately execute a request, without first establishing a total order, whereas Byzantine agreement between replicas is used, either periodically or on-demand, to establish a common state synchronization point as well as to identify the set of individual operations needed to resolve conflicts. Meanwhile, the client must wait for enough replies from a majority of replicas (after Byzantine agreement is achieved) to commit a reply, which is clearly blocking and impose high delays under network partitions or high latency. Another major issue is that servers may stop receiving new requests until Byzantine agreement among servers is achieved to withstand a Byzantine client. We believe that this is impractical in scenarios where eventual consistency was selected to not forfeit availability.

The proposed protocol In this paper a protocol that makes eventual consistency "as secure as possible", without impact on system's availability nor requiring a significant modification to an already deployed system. The protocol allows the service to run in an eventually consistent manner whereas Byzantine behaviors are detected off the critical path, in a back-end process, with the help of a black-box BFT cluster. In particular, and as described in Fig. 8.2.1, client's requests are served by an associated application server as they arrive without immediate synchronization with other servers, which is done in the background and eventually leading to data convergence. Decoupled form this front-end logic, a server progressively sends consistent data offsets to the BFT cluster to be matched against similar versions of other servers, thus forming a "certificate": a signed proof that up to this very offset, data is equivalent on an appropriate majority of non Byzantine application servers. The client progressively receives the most recent certificate along

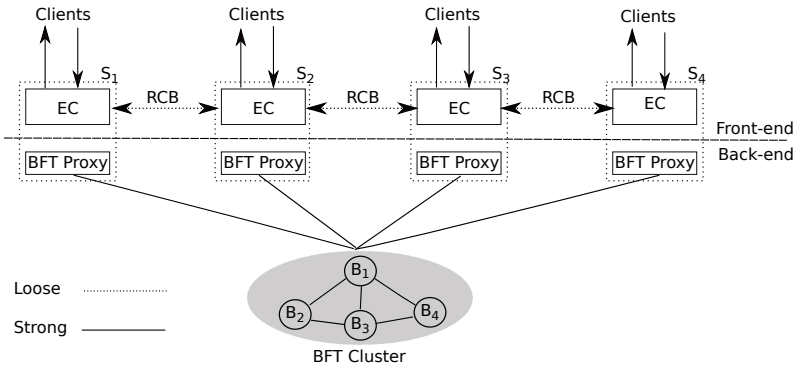


Figure 8.2.1: The system model showing how a consistent offset is always verified through the BFT cluster (back-end) without hindering clients access to application servers S_j (front-end) through eventual consistency. S_i are loosely coupled via a Reliable Causal Broadcast (RCB).

the replies of the associated server. This allows the client to verify the validity of the certificate; otherwise, it may switch to another server if it holds a proof (basically an invalid certificate) of detecting a Byzantine server, or if the certificate is not sufficiently up to date (which is verified through the other servers as well).

Discussion and future work One may argue that our solution is not sufficiently secure as clients can receive non certified data. While this is true, the client will be able to progressively detect any misbehaviors once the consistent data offset evolves. In our opinion, adopting more secure solutions like fault prevention or hiding will impose extra delays as it is done in the critical path, whereas our solution is accountable for Byzantine faults without impacting availability. We believe that in the same sense that the adopters of EC trade strong consistency — despite being a correctness property — for availability, they will likely be keen to trade high security in favor of high availability. What supports our argument is that current EC solutions in production still run in the wild without such Byzantine guarantees; and therefore, they may be less reluctant to adopt secure solutions like ours provided that availability is not compromised.

The solution we introduce is interesting for both: service and applications. On the service side, our solution is important as it guarantees convergence despite the presence of Byzantine servers or clients, which is not possible in current EC systems. On the application side, it is interesting due to its flexibility through allowing a spectrum of options: A non sensitive client can proceed with operations without checking the certificate (i.e., as current systems do), whereas a very conservative client can only accept read operations from a certified consistent data (on the expense of stale data); a trade-off option is to accept a limited number of operations ahead the certified data as long as they will be verified in the future and can be rolled back.

We describe a short version of the protocol in [93]; while at the meantime, we are implementing a prototype of the protocol to perform an empirical evaluation to assess the usefulness and feasibility of our approach.

Chapter 9

Annotated Publications & Dissemination

9.1 Publications

We present a list of the scientific papers and reports where the work towards D3.1 has been presented:

- Nuno Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types. To appear in Encyclopedia of Big Data Technologies, 2018.

- Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. CoRR, abs/1710.04469, 2017.
  Abstract
  Distributed systems designed to serve clients across the world often make use of geo-replication to attain low latency and high availability. Conflict-free Replicated Data Types (CRDTs) allow the design of predictable multi-master replication and support eventual consistency of replicas that are allowed to transiently diverge. CRDTs come in two flavors: state-based, where a state is changed locally and shipped and merged into other replicas; operation-based, where operations are issued locally and reliably causal broadcast to all other replicas. However, the standard definition of op-based CRDTs is very encompassing, allowing even sending the full-state, and thus imposing storage and dissemination overheads as well as blurring the distinction from state-based CRDTs. We introduce pure op-based CRDTs, that can only send operations to other replicas, drawing a clear distinction from state-based ones. Data types with commutative operations can be trivially implemented as pure op-based CRDTs using standard reliable causal delivery; whereas data types having non-commutative operations are implemented using a PO-Log, a partially ordered log of operations, and making use of an extended API, i.e., a Tagged Causal Stable Broadcast (TCSB), that provides extra causality information upon delivery and later informs when delivered messages become causally stable, allowing further PO-Log compaction. The framework is illustrated by a catalog of pure op-based specifications for classic CRDTs, including counters, multi-value registers, add-wins and remove-wins sets.

- Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Saturn: A Distributed Meta-

data Service for Causal Consistency. EuroSys, 2017.
Abstract
This paper presents the design, implementation, and evaluation of Saturn, a metadata service for geo-replicated systems. Saturn can be used in combination with several distributed and replicated data services to ensure that remote operations are made visible in an order that respects causality, a requirement central to many consistency criteria. Saturn addresses two key unsolved problems inherent to previous approaches. First, it eliminates the tradeoff between throughput and data freshness, when deciding what metadata to use for tracking causality. Second, it enables genuine partial replication, a key property to ensure scalability when the number of geo-locations increases. Saturn addresses these challenges while keeping metadata size constant, independently of the number of clients, servers, data partitions, and locations. By decoupling metadata management from data dissemination, and by using clever metadata propagation techniques, it ensures that the throughput and visibility latency of updates on a given item are (mostly) shielded from operations on other items or locations. We evaluate Saturn in Amazon EC2 using realistic benchmarks under both full and partial geo-replication. Results show that weakly consistent datastores can lean on Saturn to upgrade their consistency guarantees to causal consistency with a negligible penalty on performance.

- Gonçalo Cabrita and Nuno M. Preguiça. Non-uniform replication. In *Proceedings of OPODIS 2017*, 2017.
Abstract
Replication is a key technique in the design of efficient and reliable distributed systems. As information grows, it becomes difficult or even impossible to store all information at every replica. A common approach to deal with this problem is to rely on partial replication, where each replica maintains only a part of the total system information. As a consequence, a remote replica might need to be contacted for computing the reply to some given query, which leads to high latency costs particularly in geo-replicated settings. In this work, we introduce the concept of nonuniform replication, where each replica stores only part of the information, but where all replicas store enough information to answer every query. We apply this concept to eventual consistency and conflict-free replicated data types. We show that this model can address useful problems and present two data types that solve such problems. Our evaluation shows that non-uniform replication is more efficient than traditional replication, using less storage space and network bandwidth.

- Vitor Enes. Efficient Synchronization of State-based CRDTs. Masters thesis. Universidade do Minho. 2017.
Abstract
To ensure high availability in large scale distributed systems, Conflict-free Replicated Data Types (CRDTs) relax consistency by allowing immediate query and update operations at the local replica, with no need for remote synchronization. State-based CRDTs synchronize replicas by periodically sending their full state to other replicas, which can become extremely costly as the CRDT state grows. Delta-based CRDTs address this problem by producing small incremental states (deltas) to be used in synchronization instead of the full state. However, current synchronisation algorithms for Delta-based CRDTs induce redundant wasteful delta propagation,

performing worse than expected, and surprisingly, no better than State-based. In this paper we: 1) identify two sources of inefficiency in current synchronization algorithms for delta-based CRDTs; 2) bring the concept of join decomposition to state-based CRDTs; 3) exploit join decompositions to obtain optimal deltas and 4) improve the efficiency of synchronization algorithms; and finally, 5) evaluate the improved algorithms.

- Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta State Replicated Data Types. Journal of Parallel Distributed Computing. 2018.
  Abstract
  Conflict-free Replicated Data Types (CRDTs) are distributed data types that make eventual consistency of a distributed object possible and non ad-hoc. Specifically, state-based CRDTs ensure convergence through disseminating the entire state, that may be large, and merging it to other replicas. We introduce Delta State Conflict-Free Replicated Data Types ($\delta - CRDT$) that can achieve the best of both operation-based and state-based CRDTs: small messages with an incremental nature, as in operation-based CRDTs, disseminated over unreliable communication channels, as in traditional state-based CRDTs. This is achieved by defining $\delta - mutators$ to return a *delta-state*, typically with a much smaller size than the full state, that to be joined with both local and remote states. We introduce the $\delta - CRDT$ framework, and we explain it through establishing a correspondence to current state-based CRDTs. In addition, we present an anti-entropy algorithm for eventual convergence, and another one that ensures causal consistency. Finally, we introduce several $\delta - CRDT$ specifications of both well-known replicated datatypes and novel datatypes, including a generic map composition.

- Paulo Sérgio Almeida and Carlos Baquero. Scalable eventually consistent counters over unreliable networks. Journal of Distributed Computing. 2018.
  Abstract
  Counters are an important abstraction in distributed computing, and play a central role in large scale geo-replicated systems, counting events such as web page impressions or social network "likes". Classic distributed counters, strongly consistent via linearisability or sequential consistency, cannot be made both available and partition-tolerant, due to the CAP Theorem, being unsuitable to large scale scenarios. This paper defines Eventually Consistent Distributed Counters (ECDCs) and presents an implementation of the concept, Handoff Counters, that is scalable and works over unreliable networks. By giving up the total operation ordering in classic distributed counters, ECDC implementations can be made AP in the CAP design space, while retaining the essence of counting. Handoff Counters are the first Conflict-free Replicated Data Type (CRDT) based mechanism that overcomes the identity explosion problem in naive CRDTs, such as G-Counters (where state size is linear in the number of independent actors that ever incremented the counter), by managing identities towards avoiding global propagation and garbage collecting temporary entries. The approach used in Handoff Counters is not restricted to counters, being more generally applicable to other data types with associative and commutative operations.

- Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and João Leitão. Borrowing

an Identity for a Distributed Counter. PaPoC'17, 2017.
Abstract Conflict-free Replicated Data Types (CRDTs) are data abstractions (registers, counters, sets, maps, among others) that provide a relaxed consistency model called Eventual Consistency. Current designs for CRDT counters do not scale, having a size linear with the number of both active and retired nodes (i.e., nodes that leave the system permanently after previously manipulating the value of the counter). In this paper we present a new counter design called Borrow-Counter, that provides a mechanism for the retirement of transient nodes, keeping the size of the counter linear with the number of active nodes.

- Ali Shoker, Houssam Yactine, and Carlos Baquero. As Secure As Possible Eventual Consistency. PaPoC '17, 2017.
Abstract
Eventual consistency (EC) is a relaxed data consistency model that, driven by the CAP theorem, trades prompt consistency for high availability. Although, this model has shown to be promising and greatly adopted by industry, the state of the art only assumes that replicas can crash and recover. However, a Byzantine replica (i.e., arbitrary or malicious) can hamper the eventual convergence of replicas to a global consistent state, thus compromising the entire service. Classical BFT state machine replication protocols cannot solve this problem due to the blocking nature of consensus, something at odd with the availability via replica divergence in the EC model. In this work in progress paper, we introduce a new secure highly available protocol for the EC model that assumes a fraction of replicas and any client can be Byzantine. To respect the essence of EC, the protocol gives priority to high availability, and thus Byzantine detection is performed off the critical path on a consistent data offset. The paper concisely explains the protocol and discusses its feasibility. We aim at presenting a more comprehensive and empirical study in the future.

- Vitor Enes, Paulo Sérgio Almeida and Carlos Baquero. The Single-Writer Principle in CRDT Composition. PMLDC, ECOOP, 2017.
Abstract
Multi-master replication in a distributed system setting allows each node holding a replica to update and query the local replica, and disseminate updates to other nodes. Obtaining high availability typically entails allowing replicas to diverge and requires a background mechanism for re-establishing consistency. Conflict-free Replicated Data Types (CRDTs) extend standard sequential data-types with appropriate merge functions, and often can be composed together to create more complex ones. In this work we add a generic CRDT composition approach that explores the single-writer principle. By carefully controlling which part of the composition can be updated by each replica, we can derive efficient designs that cover new use-cases. After introducing the new construction we exemplify some uses, including how to emulate a simple Doodle functionality for selecting a common meeting schedule among different participants.

- Georges Younes, Paulo Sérgio Almeida and Carlos Baquero. Compact Resettable Counters through Causal Stability. PaPoC, 2017.
Abstract

Conflict-free Data Types (CRDTs) were designed to automatically resolve conflicts in eventually consistent systems. Different CRDTs were designed in both operation-based and state-based flavors such as Counters, Sets, Registers, Maps, etc. In a previous paper [2], Baquero et al. presented the problem with embedded CRDT counters and a solution, covering state-based counters that can be embedded in maps, but needing an ad-hoc extension to the standard counter API. Here, we present a resettable operation-based counter design, with the standard simple API and small state, through a causal-stability-based state compaction.

9.2 Dissemination

- Peer Stritzinger. *Fixing Erlang's Distribution Protocol*. Erlang User Conference 2017, Stockholm, Sweden, June 8-9, 2017. See: `https://youtu.be/CVpcrYH18NE`.
- Peer Stritzinger. *Robotics and Sensors Using Erlang on Embedded Systems with GRiSP*. CodeMesh 2017, London, UK, Nov. 7-9, 2017.
- Ali Shoker. *There are no BFT Fans Anymore ... About Secure Eventual Consistency*. Curry On!, Barcelona, Spain, June 2017.

Bibliography

[1] Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx.

[2] Akka Documentation: Cluster, 2017. https://doc.akka.io/docs/akka/2.5/index-cluster.html.

[3] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceeding of the IEEE 36th International Conference on Distributed Computing Systems*, ICDCS'16, pages 405–414, Nara, Japan, 2016.

[4] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. Bounded version vectors. In Rachid Guerraoui, editor, *Distributed Computing*, pages 102–116, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[5] P. S. Almeida and C. Baquero. Scalable eventually consistent counters over unreliable networks. *Distributed Computing*, 2018.

[6] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval tree clocks. In Theodore P. Baker, Alain Bui, and Sébastien Tixeuil, editors, *Principles of Distributed Systems*, pages 259–274, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta State Replicated Data Types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018.

[8] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018.

[9] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proc. 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013.

[10] Gustavo Alonso. Partial database replication and group communication primitives. In *Proc. European Research Seminar on Advances in Distributed Systems*, 1997.

[11] Amazon. Amazon IoT Greengrass reference architecture, 2018. https://aws.amazon.com/greengrass/.

[12] Yair Amir, Danny Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*, pages 76–84, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[13] Yair Amir and Jonathan Stanton. The spread wide area group communication system. 2007.

[14] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In Laurent Réveillère, Tim Harris, and Maurice Herlihy, editors, *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 6:1–6:16. ACM, 2015.

[15] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno M. Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS 2015, Montreal, QC, Canada, September 28 - October 1, 2015*, pages 31–36. IEEE Computer Society, 2015.

[16] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 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*, pages 126–140, 2014.

[17] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. *CoRR*, abs/1710.04469, 2017.

[18] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991.

[19] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.

[20] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.

[21] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proc. of MCC*, 2012.

[22] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Saturn: a distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 111–126. ACM, 2017.

[23] Eric Brewer. On a certain freedom: exploring the CAP space. Invited talk at PODC 2010, Zurich, Switzerland, July 2010.

[24] J. Byers, J. Considine, and M. Mitzenmacher. Fast Approximate Reconciliation of Set Differences. Technical report, CS Dept., Boston University, 2002.

[25] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.

[26] Gonçalo Cabrita and Nuno M. Preguiça. Non-Uniform Replication. In *Proceedings of the 21st International Conference on Principles of Distributed Systems, OPODIS 2017*, pages 24:1–24:19. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, December 2017.

[27] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[28] Hua Chai and Wenbing Zhao. Byzantine fault tolerance for services with commutative operations. In *Proceedings of the 2014 IEEE International Conference on Services Computing*, SCC '14, pages 219–226, Washington, DC, USA, 2014. IEEE Computer Society.

[29] Punit Chandra, Pranav Gambhire, and Ajay Kshemkalyani. Performance of the optimal causal multicast algorithm: A statistical analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 15:40– 52, 02 2004.

[30] Natalia Chechina, Kenneth MacKenzie, Simon Thompson, Phil Trinder, Olivier Boudeville, Viktória Fördős, Csaba Hoch, Amir Ghaffari, and Mario Moro Hernandez. Evaluating scalable distributed erlang for scalability and reliability. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2244–2257, 2017.

[31] Natalia Chechina, Phil Trinder, Amir Ghaffari, Rickard Green, Kenneth Lundin, and Robert Virding. The design of scalable distributed erlang. In *DRAFT PROCEEDINGS OF THE 24TH SYMPOSIUM ON IMPLEMENTATION AND APPLICATION OF FUNCTIONAL LANGUAGES (IFL 2012)*, page 461.

[32] Edge Computing Consortium and Alliance of Industrial Internet. ECC Reference Architecture, 2018. http://en.ecconsortium.net/Uploads/file/20180328/1522232376480704.pdf.

[33] Open Fog Consortium. OpenFog Reference Architecture, 2017. https://www.openfogconsortium.org/wp-content/uploads/OpenFog_Reference_Architecture_2_09_17-FINAL.pdf.

[34] The Syncfree Consortium. FP7 SyncFree Project, 2017. https://syncfree.lip6.fr.

[35] The Syncfree Consortium. Antidote db, 2018. http://syncfree.github.io/antidote/.

[36] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-distributed

Database. In *Proc. 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[37] Tyler Crain and Marc Shapiro. Designing a Causally Consistent Protocol for Geo-distributed Partial Replication. In *Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, 2015.

[38] Nigel Davies, Nina Taft, Mahadev Satyanarayanan, Sarah Clinch, and Brandon Amos. Privacy Mediators: Helping IoT Cross the Chasm. In *Proc. of HotMobile*, 2016.

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

[40] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *PODC*, 1987.

[41] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, 1987.

[42] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the 5th ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, Seattle, WA, USA, 2014.

[43] Thomas Eisenbarth and Sandeep Kumar. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 24(6), 2007.

[44] Vitor Enes. Efficient Synchronization of State-based CRDTs. Master's thesis, Universidade do Minho, 2017.

[45] Vitor Enes, Paulo Sérgio Almeida, and Carlos Baquero. The Single-Writer Principle in CRDT Composition. In *Second Workshop on Programming Models and Languages for Distributed Computing, PMLDC@ECOOP 2017, Barcelona, Spain, June 20*, 2017.

[46] Vitor Enes, Carlos Baquero, Paulo Sérgio Almeida, and João Leitão. Borrowing an Identity for a Distributed Counter: Work in Progress Report. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC'17, pages 4:1–4:3. ACM, 2017.

[47] Daniel Engels, Xinxin Fan, Guang Gong, Honggang Hu, and Eric M Smith. Hummingbird: ultra-lightweight cryptography for resource-constrained devices. In *International Conference on Financial Cryptography and Data Security*, pages 3–18. Springer, 2010.

[48] ETSI. MEC Reference Architecture, 2018. https://www.etsi.org/.

[49] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proc. of USENIX Security*, 2016.

[50] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[51] Linux Foundation. EdgeX Foundry Reference Architecture, 2018. https://docs.edgexfoundry.org/Ch-Architecture.html.

[52] Amir Ghaffari. Investigating the scalability limits of distributed erlang. In *Proceedings of the Thirteenth ACM SIGPLAN workshop on Erlang*, pages 43–49. ACM, 2014.

[53] Amir Ghaffari, Natalia Chechina, Phil Trinder, and Jon Meredith. Scalable persistent storage for erlang: Theory and practice. In *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*, pages 73–74. ACM, 2013.

[54] 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.

[55] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review*, 39(1):68–73, 2008.

[56] JGroups. a toolkit for reliable multicast communication, 2002.

[57] Ajay D. Kshemkalyani and Mukesh Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distrib. Comput.*, 11(2):91–111, April 1998.

[58] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, volume 8878 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2014.

[59] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, November 1992.

[60] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8), 1974.

[61] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[62] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[63] João Leitão, José Pereira, and Luis Rodrigues. HyParView: A membership protocol for reliable gossip-based broadcast. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 419–429. IEEE, 2007.

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

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

[66] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, Cascais, Portugal, 2011.

[67] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 313–328, 2013.

[68] Hewlett Packard Enterprise Development LP. HPE Edge Center. https://www.hpe.com/us/en/product-catalog/detail/pip.hpe-micro-datacenter.1009483818.html.

[69] Josh Macdonald. Xdelta.

[70] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.*, 6(9), July 2013.

[71] Friedemann Mattern. Virtual time and global states of distributed systems. In *PARALLEL AND DISTRIBUTED ALGORITHMS*, pages 215–226. North-Holland, 1988.

[72] Kerry A McKay, Larry Bassham, Meltem Sönmez Turan, and Nicky Mouha. Report on lightweight cryptography. *NIST DRAFT NISTIR*, 8114, 2016.

[73] Christopher Meiklejohn and Peter Van Roy. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 184–195. ACM, 2015.

[74] Hein Meling, Alberto Montresor, Bjarne E. Helvik, and Ozalp Babaoglu. Jgroup/arm: a distributed object group platform with autonomous replication management. *Software: Practice and Experience*, 38(9):885–923.

[75] Microsoft. Azure IoT reference architecture, 2018. http://download.microsoft.com/download/A/4/D/A4DAD253-BC21-41D3-B9D9-87D2AE6F0719/Microsoft_Azure_IoT_Reference_Architecture.pdf.

[76] Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Blotter: Low Latency Transactions for Geo-Replicated Storage. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 263–272, 2017.

[77] Achour MostÃ©faoui and M Raynal. Causal multicasts in overlapping groups: towards a low cost approach. pages 136 – 142, 10 1993.

[78] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. ACM.

[79] David Navalho, Sérgio Duarte, and Nuno Preguiça. A Study of CRDTs that do Computations. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15. ACM, 2015.

[80] Patrick E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986.

[81] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 1989.

[82] Ravi Prakash, Michel Raynal, and Mukesh Singhal. An adaptive causal ordering algorithm suited to mobile computing environments. *J. Parallel Distrib. Comput.*, 41(2):190–204, March 1997.

[83] Nuno M. Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *33rd IEEE International Symposium on Reliable Distributed Systems Workshops, SRDS Workshops 2014, Nara, Japan, October 6-9, 2014*, pages 30–33. IEEE Computer Society, 2014.

[84] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types. To appear in Encyclopedia of Big Data Technologies.

[85] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, October 1991.

[86] L. Rodrigues, J. Pereira, and J. Leitao. Epidemic broadcast trees. In *Reliable Distributed Systems, IEEE Symposium on(SRDS)*, volume 00, pages 301–310, 10 2007.

[87] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1), March 2005.

[88] André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 219–232, London, UK, UK, 1989. Springer-Verlag.

[89] André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 219–232, London, UK, UK, 1989. Springer-Verlag.

[90] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 2010 IEEE 29th International Symposium on Reliable Distributed Systems*, SRDS '10, pages 214–224, New Delhi, Punjab, India, 2010.

[91] Nicolas Schiper and Fernando Pedone. Fast, flexible, and highly resilient genuine fifo and causal multicast algorithms. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 418–422, New York, NY, USA, 2010. ACM.

[92] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[93] Ali Shoker, Houssam Yactine, and Carlos Baquero. As secure as possible eventual consistency: Work in progress. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '17, pages 5:1–5:5, New York, NY, USA, 2017. ACM.

[94] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Inf. Process. Lett.*, 43(1):47–52, August 1992.

[95] Delta Power Solutions. Micro Data Center. http://www.deltapowersolutions.com/en/mcis/data-center-solutions-micro-datacenter.php.

[96] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[97] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, 2011.

[98] Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. *Advances in Cryptology-ASIACRYPT 2010*, pages 377–394, 2010.

[99] Pat Stephenson and Kenneth Birman. Fast causal multicast. *SIGOPS Oper. Syst. Rev.*, 25(2):75–79, April 1991.

[100] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS) 94, Austin, Texas, September 28-30, 1994*, pages 140–149. IEEE Computer Society, 1994.

[101] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proc. 24th ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[102] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical report, Australian National University, 1998.

[103] Robbert van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. Efficient Reconciliation and Flow Control for Anti-entropy Protocols. In *LADIS*, 2008.

[104] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. *SIGOPS Oper. Syst. Rev.*, 41(6):59–72, October 2007.

[105] Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1), January 2009.

[106] Claes Wikström. Distributed programming in erlang. In *the 1st International Symposium on Parallel Symbolic Computation (PASCO 94)*, pages 412–421. World Scientific, 1994.

[107] Georges Younes, Paulo Sérgio Almeida, and Carlos Baquero. Compact resettable counters through causal stability. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '17, pages 2:1–2:3, New York, NY, USA, 2017. ACM.

[108] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a Trillion (Unfixable) Flaws on a Billion Devices: Rethinking Network Security for the Internet-of-Things. In *Proc. of HotNets*, 2015.

[109] Igor Zavalyshyn, Nuno O Duarte, and Nuno Santos. Homepad: Guardian of a smart home galaxy.

[110] Marek Zawirski, Carlos Baquero, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Eventually consistent register revisited. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, pages 9:1–9:3, New York, NY, USA, 2016. ACM.

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