LIGHTKONE

Lightweight computation for networks at the edge

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

D5.1: Infrastructure Support for Aggregation in Edge Computing

Deliverable no.: D5.1
Title: Infrastructure Support for Aggregation in Edge Computing
Due date of deliverable: December 31, 2017
Actual submission date: February 9, 2018

Lead contributor: NOVA
Revision: 1.0
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
01/12/2017	0.1	1st draft with outline and ToC	NOVA
29/01/2018	0.5	1st complete draft of the deliverable	NOVA
09/02/2018	1.0	Deliverable is complete and ready for submission	NOVA

Contributors:

Contributor	Institution
João Leitão	NOVA
Nuno Preguiça	NOVA
Henrique Domingos	NOVA
Sérgio Duarte	NOVA
Pedro Ákos Costa	NOVA
Pedro Fouto	NOVA
Guilherme Borges	NOVA
Vasileios Karagiannis	NOVA
Peter Van Roy	UCL
Christopher Meiklejohn	UCL & IST/INESC-ID
Roger Pueyo Centelles	UPC
Carlos Baquero	INESC TEC
Giorgos Kostopoulos	GLUK

Contents

1 Executive summary

Moving computations towards the edge of distributed systems is an essential endeavor that ensures the continued growth and sustainability of large-scale distributed applications. Doing so, also enables the emergence of novel applications that leverage on the edge computing paradigm and offer better and improved services to end users. We refer to such applications as edge-enabled applications.

The Lightkone consortium posits itself as a leading force in the development of new techniques and support to enable this new generation of edge-enabled applications. We expect edge-enabled applications to have components executing over a broad spectrum of edge scenarios. These scenarios range from data centers and nodes located in their vicinity (what we refer to as *heavy edge*), to locations closer to end users, or even in their devices (what we also refer to as *light edge*). One of the end goals of the Lightkone project is to develop new methodologies, solutions, and tools that enable application developers to fully tap into the potential of the edge computing paradigm. Hence, new solutions will have to be developed to tackle challenges that arise at multiple points of this spectrum.

This report discusses the first steps of the Lightkone consortium in addressing the multitude of challenges related to the materialization of edge computing solutions for the light edge-end of the spectrum. In particular, challenges related to the inherent complexities of developing and executing available and correct edge-enabled applications with few dependencies on the core of the system (i.e, heavy edge). The contributions reported here are presented in two groups. The first, which we call *main contributions*, provides mechanisms and tools to ease the development and support the execution of distributed protocols and applications in light edge scenarios. The second, that we name *secondary contributions*, delves into solutions and support for improving the collaboration among multiple light edge components and their interactions with the heavy edge (e.g., services running in cloud or fog scenarios).

The main contributions presented in this deliverable can be summarized as follows:

Yggdrasil and Aggregation Protocols for Wireless Settings: Yggdrasil is a novel framework, written in the C programming language to simplify the task of designing distributed protocols and applications for general purpose ad hoc networking. Yggdrasil enables applications to specify the type of ad hoc wireless network they require through an intuitive API which hides the complexity of configuring the wireless radios manually. Additionally, the framework implements basic communication mechanisms to simplify the exchange of messages among multple participating devices (e.g., a mac layer broadcasat technique). We use Yggdrasil to build a set of simple membership and aggregation protocols, to showcase the usability of the framework. The framework will be used both for experimental comparison of such protocols (using out-of-shelf hardware such as raspberry pis) while also offering the potential to be used as a teaching tool, by enabling university students to have access to a practical framework that simplifies the development and testing of ad hoc protocols.

Edge computing with the Legion Framework: Edge computing addresses schemes for performing computations outside the data center boundaries. One possible materialization of edge computing pushes computations towards the end-user devices.

Legion, is a framework that supports computation at the edge for web applications, by leveraging on the user browsers to perform such computations. At its core, Legion is a framework written in javascript, that leverages WebRTC and HTML5 primitives, to allow web application clients to replicate portions of the web application state, enabling its manipulation directly at the browser, potentially disconnected from the server. Since Legion targets collaborative web applications, the relevant application state replicated by Legion is encapsulated in CRDTs, which can then be synchronized directly among clients manipulating the same portions of the application state. Furthermore, Legion also offers the possibility to interconnect with existing cloud platforms (such as the Google Real Time API) and offers simple security mechanisms to avoid the manipulation of application state by non-authorized users.

Edge computing with the LASP Framework: Another possible materialization of edge computing resorts to private grids and cloud data centers (or small/regional grids and clouds) to perform computations outside the boundaries of the centralized (main) data centers, and closer to clients. LASP is a framework, written in erlang, that provides primitives for performing computations over data in a synchronization free way at these edge scenarios. LASP resorts to CRDTs to model the data being manipulated, and provides a set of operators that enable to express computations, across large numbers of processes as transformations among different CRDT types, that might reside and be replicated at different nodes of the system.

GRiSP Platform and Hardware: Edge computing entails high acquisition of data in the edge by specialized sensor devices with networking capabilities and low energy consumption. Ideally, such devices will have some computing power to enable some edge computations. To make some specific edge computing applications efficient and reliable, these data acquisition devices must evolve as to cope with the high demands of emerging applications, both in terms of load but also in terms of reliability. To explore this venue, we also present the design of a new integrated system, comprising a sensor rich hardware module that natively boots an Erlang VM. This was designed in the context of the Lightkone efforts to push forward the boundaries of current edge computing environments.

The secondary contributions presented in this report are as follows:

Non-Uniform CRDTs: We present a novel contribution related with the design and implementation of a variant of Conflict-free Replicated Data Types (CRDTs) that we denote Non-Uniform CRDTs. This abstraction follows the original proposal of CRDTs, that allow an application to encode its state into data structures that can be freely replicated and manipulated in a total independent fashion (i.e, without any form of synchronization). CRDTs synchronize in background, and their internal operation allows the multiple replicas to converge to a single state, that depends on the intended semantics of the application. Non-Uniform CRDTs extend this design to enable CRDTs to synchronize only a fraction of their internal state, while enabling computations performed over these CRDTs to achieve the same result. This can be leveraged to compute aggregation functions efficiently over large collections of data. Moreover, Non-Uniform CRDTs point towards a concrete venue

in developing an efficient data encoding strategy that can be leveraged to support the interaction between different components of edge-enabled applications. In particular, to enable inter-operation between components executing on distinct edge scenarios and execution environments.

Enriching Consistency at the Edge: Some edge computing applications are expected to have edge nodes accessing large quantities of data that might reside in different components of the system (potentially in the cloud) in an efficient and reliable way. Due to the necessity to replicate data, even in the context of cloud computing for fault tolerance and scalability, and the need to ensure that systems are always available, data might be exposed to clients with weak consistency guarantees. This makes the life of application developers hard, as they have to cope with consistency anomalies in their applications logic. This can be exacerbated when using third-party and legacy systems. To assist in this, we present a novel contribution that enables clients to enrich the consistency properties exposed by replicated systems up to causal consistency. This solution operates through a middleware that intercepts, transparently, the calls executed by the local client to an external (storage) system. This middleware modifies the returned values given the history of that client, and small metadata hints stored in the system. We believe this form of abstraction can be useful to simplify the development of complex edge enabled distributed applications, by transparently hiding anomalies that arise due to the use of weak consistency models by legacy systems.

1.1 Software Artifacts

In addition to the description of the main conceptual contributions achieved by the Lightkone consortium in the context of the Work Package 5, we also report on software artifacts that were produced. These artifacts materialize some of these contributions.

The software artifacts presented as part of this deliverable are the following: *i*) the Yggdrasil framework; *ii*) the Legion framework; *iii*) the Lasp framework; and finally, *iv*) the software to support the development of applications in the GRiSP board.

All software artifacts are available currently through the git repository of the work package.

2 Introduction

Since its inception in 2005, the cloud computing paradigm has deeply impacted the design and implementation of user-facing distributed systems [79]. Nowadays, most services resort to cloud infrastructures to store both application and user data, perform complex computations over data, and provide services for users bases that are scattered throughout the world. This paradigm has lead to a rise in popularity of social network applications and mobile applications, among others. In the particular case of mobile applications, the cloud is able to circumvent the resources limitations of mobile devices, enabling the mobile devices to only execute a thin cache and presentation layer to users, while all application logic and data are handled by cloud infrastructures [37].

The cloud computing paradigm, however, is not a panacea for building and executing reliable and efficient distributed applications. In fact, the use of cloud-based infrastructures presents significant downsides [36]. In particular, the use of cloud infrastructures has monetary costs for application operators, that not only grow with the effective amount of processing and stored data, but also with the amount of data transferred between (client) applications and the cloud infrastructure; furthermore, even cloud infrastructures have limits to their scalability and, under heavy usage, the latency experienced by users when interacting with applications can become too high. Finally, outsourcing data storage to the cloud comes with a significative risk for data privacy: on one hand, storing user data in the cloud implies that the user is releasing the control of its own data and, on the other hand, the cloud is a particularly attractive target for internet hackers due the potential high return for a successful attack, as demonstrated by incidents in the past [46, 57].

The rise in popularity of user-centric applications, such as the above mentioned social networks and mobile applications, associated with the rapid growth in popularity of the Internet-of-Things (IoT) applications has lead to a shift in the behavior and requirements of distributed applications. The most obvious of these changes is that data is now produced and consumed by clients whereas, in the past, many distributed applications had clients consuming the data that was mostly produced at the center of the system. Additionally, the volume of data produced by client devices, particularly in the context of IoT applications, has increased significantly, and is expected to continue to do so, as denoted by recent reports [28, 79]. This puts a significant burden on cloud infrastructures, that must be able to receive and store huge quantities of data generated by the clients and process it to provide useful information back to those clients, a burden that will become impossible to support in the near future [79]. This shift also exacerbates the downsides of the cloud computing paradigm discussed previously. Latency experienced by users and applications will tend to rise due to the time required to transfer large quantities of data. At the same time, in some application domains such as medical care IoT applications, the private nature of the data being manipulated by applications becomes more and more pronounced.

This has lead to the emergence of a new computing paradigm, usually called *edge computing* where some, or most, data computations over user-generated data is moved beyond the data center boundaries towards the edge of the system, closer to the end user devices that both generate the data and consume the (processed) data. Such approach reduces the load on both cloud infrastructures and networks, while allowing data to be produced for end user consumption in a more expedite way. Furthermore, such an ap-

proach has the additional advantage of avoiding the transmission of private user data to an external infrastructure that lies beyond the user control.

The practicality of edge computing however, requires a complete rethinking on how to build and execute distributed applications. Applications have to be flexible enough to deal with a large number of heterogeneous devices that support the execution of different components of these applications. Devices themselves might need to evolve as to provide the adequate hardware (and associated software stacks) required to support general purpose edge computing applications. Furthermore, deciding where data should be sent and processed has to be handled at run-time, in order to take into consideration the operational environment of applications.

Additionally, computations themselves have to be specified by application programmers in a different way, enabling computations to operate over incomplete data sets, and avoiding strong synchronization between the devices that generate data, and those that support the execution of such computations, as to avoid the creation of bottlenecks in the system and to promote fault-tolerance and availability. Finally, data has to be protected, as to avoid its manipulation by unauthorized entities and protect the privacy of sensitive data.

When speaking about edge computing, one can envision multiple edge scenarios. These scenarios can range from the use of small data centers and dedicated infrastructure that are located beyond the data center boundary, and that can be responsible for managing data generated at a national or regional level. This is a particular edge scenario commonly referred by Fog Computing [23, 94]. Going farther away from the core of the network, one can find federation of resources (i.e, servers, services, sensors, and actuators) at the scale of a city (a scenario that falls within the scope of smart cities and sensor networks) or gridlets and cloudlets of small computational devices very close to end users. Finally, one can rely on end users' devices, such as laptops, desktops, or mobile devices to perform computations (a field usually called peer-to-peer computing or mobile edge computing for the particular case of mobile devices).

One of the end goals of the Lightkone project is to develop new methodologies, solutions, and tools that enable application developers to fully tap into the potential of the edge computing paradigm. To build effective and robust applications that fully leverage the potential of the edge (i.e., edge-enabled applications), one has to be able to manipulate data in all scenarios discussed above, with a high degree of independence from the particularities of the hardware or the execution environment where these computations are performed. The most fundamental data computations that can be tackled in this context are data aggregation primitives.

Data aggregation primitives, such as average, min, max, sum, and the generation of application-dependent data-collection summaries, are an essential step to pave the way for supporting general purpose computations in the edge. This is justified by two complementary observations. First, transmitting the whole data produced at the edge among devices, even if these devices are in close proximity, will result in a performance bottleneck, as connections between devices might be limited (consider for instance devices communicating through a wireless medium in an area with significant interference). Second, data aggregation are essential primitives to build distributed monitoring services, that can extract relevant metrics concerning the availability of devices and their free resources at runtime (e.g., computational, storage, energy, etc.). Moreover, these techniques can also be used to infer details regarding the current workload of running applications. This in-

formation is essential to empower edge-enabled computing platforms to make the correct runtime planning and decisions on how to orchestrate different components and computational tasks of an edge-enabled application.

2.1 Structure of the Deliverable

The remainder of this deliverable is structured as follows:

Section 3 presents in some detail the main contributions reported as part of this deliverable.

Section 4 discusses the secondary contributions produced by the Lightkone consortium in the context of WP5.

Section 5 discusses the relationship and applicability of the main contributions reported in this deliverable with the use case applications reported previously by the Lightkone consortium in deliverable D2.1.

Section 6 discusses the relationship of the contributions reported here with the work being conducted by the Lightkone consortium in the context of other work packages.

Section 7 provides guidelines on how to access and execute the software artifacts associated with this deliverable.

Section 8 concludes this report discussing complementary research and development vectors to be pursued in the future.

3 Main Deliverable Contributions

This section presents with some detail the main contributions of this deliverable. These contributions focus on supporting the development and execution of edge-enabled applications across multiple (light) edge scenarios.

In particular this section presents the following contributions:

Section 3.1 discusses the design and implementation of the Yggdrasil framework and some of the distributed aggregation protocols developed on top of it. This contribution focuses on edge scenarios where devices communicate with each other through a wireless medium.

Section 3.2 presents the design and some of the details related to the implementation of the Legion framework. Legion, contrary to the previous contribution, focuses on an edge-scenario closely related to peer-to-peer systems. In particular the focus of Legion is on supporting edge-enabled web applications running in end-users browsers.

Section 3.3 presents the design and implementation of Lasp. Lasp is a programming abstraction and runtime support for edge applications that operate over CRDTs in fog edge computing scenarios.

Section 3.4 introduces the GRiSP platform and associated software stack. GRiSP is a novel embedded system that has the capability of running native Erlang applications. GRiSP offers a new test-bed for the design and testing of edge-enabled applications running on specialized hardware.

3.1 Yggdrasil framework and Aggregation in Wireless Settings

A particularly challenging edge scenario can be found in systems that operate at the very edge of the system, with very limited access to infrastructure network (i.e, Internet and cloud resources), where devices have to interact through wireless communication directly among them.

These edge scenarios include sensor networks, medium to large-scale Internet of Things (IoT) deployments, and some aspects of smart cities. In all of these settings, we expect to have a significant number of resource constrained nodes, that acquire large volumes of data. These devices, while being cheap to produce, and consuming low amounts of energy, have limited computational power and memory, and might be complex to program from the standpoint of application developers. For instance, many sensors have a single execution environment, where the operative system is a library linked to the application, and only a single application executes in the device.

To overcome these limitations, and to pave the way for performing general purpose computations in such edge scenarios we propose to leverage on more general purpose devices that: *i*) have more computational power and memory; *ii*) have an execution environment that is more familiar for developers; and *iii*) are affordable. One such device is the Raspberry Pi [6] whose latest model (Raspberry Pi 3 Model B) can be purchased by approximately 30 euros. This model has a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, wireless capabilities built-in, and 1Gb of RAM.

Using these devices we can enrich applications, by having some devices in the system that own additional resources. Sensors and IoT devices can connect to one of these devices to report acquired data, and these devices can locally perform computations and exchange information among them. Furthermore, these devices can also control actuators that act on their deployment environment based on the results of local computations.

To ease the deployment of such devices and to ensure scalable management, applications should be able to operate with minimal human interaction. This requires such devices to self-manage and to interact without the need of a per-device configuration. In particular these devices should be able to be added into a deployment using a plug-and-play approach, enabling systems to grow organically as required by application operators. To this end we have decided to exploit AdHoc Networking, that while avoiding the overhead of Mesh Networks, enable devices to discover each other and interact through wireless communication with no configuration.

Unfortunately, programming distributed applications in this setting is a non-trivial. There are two main reasons for this; first, AdHoc Networking requires WiFi radios of devices to be configured appropriately. Furthermore, in some cases, as for instance high density scenarios, it could be useful to enable devices to switch (at run time) among multiple AdHoc Networks, operating under different frequencies, as to avoid frequent collisions in the Wireless medium, which would significantly degrade the operation of applications, or render them completely ineffective. The second reason, is that there are few frameworks that support the construction of distributed protocols in AdHoc sce-

narios, Such frameworks should provide to programmers abstractions to send and receive messages through one hop broadcast directly at the MAC layer, and fundamental abstractions that are used frequently in the construction of distributed protocols and applications (such as Timer management, inter-protocol communication, and so on).

To address these challenges we have designed and implemented a prototype entitled Yggdrasil. Yggdrasil is a framework to support the development and execution of distributed protocols and applications in Wireless AdHoc networks. In this Section, we discuss the system model assumed by Yggdrasil (§ 3.1 (a)), discuss the framework design and implementation, detailing the design and implementation of a set of support protocols that are a part of Yggdrasil. We also explain how these protocols can simplify the design and implementation of both other protocols and applications (§ 3.1 (b)). We further discuss how we leveraged Yggdrasil to implement some variants of data aggregation protocols (§ 3.1 (c)), which are fundamental building blocks that pave the way to support general-purpose computations in the edge; briefly discuss the state of art (§ 3.1 (d)); and the current limitations and future plans for the framework (§ 3.1 (e)).

(a) System Model and Requirements

Yggdrasil operates at the MAC layer. It was designed for devices with, at least, one WiFi radio, capable of operating in AdHoc mode (sometimes called IBSS). It further assumes that devices have a limited amount of RAM and some processing power and can execute multi-threaded applications.

Yggdrasil further assumes that the operative system is based on linux, and in particular it assumes the availability of an implementation of the Netlink Protocol Library Suite (libnl) [3]. Library libnl exposes a set of API calls that allow to configure the wireless radio interfaces of devices. Decoupling our current design and implementation of the libnl is in our long term plans. Such decoupling will allow to execute Yggdrasil in other types of devices, such as embedded boards with wireless capabilities, as the GRiSP board described further ahead in this deliverable (§ 3.4).

(b) Yggdrasil Architecture

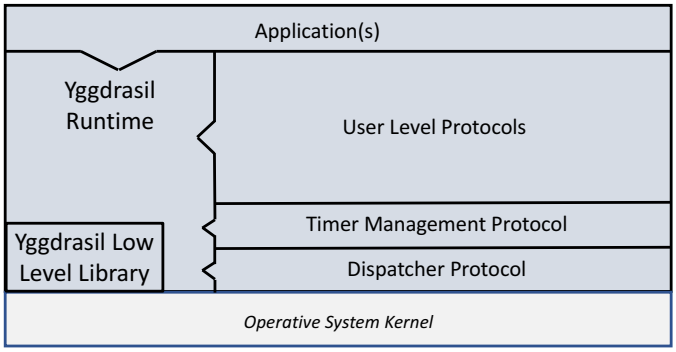


Figure 3.1: Yggdrasil Architecture Overview

We now present and discuss the Yggdrasil architecture that is illustrated in Figure 3.1. Yggdrasil allows for multiple applications to execute in a single node, resorting to a single

instance of Yggdrasil. In fact, we promote a single process model where both the Yggdrasil runtime, protocols, and applications are compiled into a single binary and hence, are executed within the context of a single process. The motivation for this is to simplify, in the future, the support of the execution of Yggdrasil applications in devices that have a single process execution model such as the GRiSP board, and multiple comercially available sensors and motes [42].

In Yggdrasil, each component (with the exception of the Yggdrasil Low Level Library and the runtime) has its own execution thread. This is valid for each of the essential support protocols (Timer Management and Dispatcher Protocol) and well as each user Protocol and Application. While this might lead to a significant number of threads for applications that resort to a large number of protocols, we made this decision to promote parallel and independent execution of protocols, as to minimize latency and avoid starvation of execution when complex computations are executed, for instance, by some client application.

Yggdrasil promotes an event driven execution environment for both protocols and applications. This is an execution model where both protocols and applications state evolve locally through the processing of asynchronous events. Events can be the reception of (network) messages, the expiration of a timer, or the reception of a event, request, or reply issued by another protocol/application in the local process. This execution model has been used in other frameworks that were designed to support the implementation and execution of distributed protocols, such as Appia [64], as it simplifies the reasoning on the operation of a distributed protocol. Overall this model leads to a more structured design of protocols. The interested reader can find more details regarding this model in [47].

In short, the main components in the Yggdrasil framework are as follows:

Yggdrasil Low Level Library: This component interacts with the kernel through the libnl, ioctl, and sockets interfaces exposed by the kernel to manage the radio interface or interfaces, configure relevant network properties, and providing low level primitives for receiving and sending messages at the MAC layer. This component exposes these functionalities through a *Channel* abstraction, that allows other components to configure, at runtime, the relevant properties of the AdHoc network being used, such as its name and radio frequency, among others. It further exposes mechanisms to enable an application to verify the networks that are enabled in the vicinity of a device, as well as mechanisms to gather information related with the wireless interfaces of the device.

Another important feature of this library, is that it allows the user to define a message pattern to filter messages received from the network at the kernel level. In the context of AdHoc Networking, one typically exchange messages at the MAC level (which allows for instance to perform one-hop broadcast [87]). However, this implies that whenever a device transmits a message in the radio frequency being employed by our AdHoc Network, Yggdrasil would receive such message. This mechanism, allows all messages sent by applications and protocols running within our framework to be tagged with a particular pattern of bytes in the header[1], enabling the framework to ignore all messages that do not exhibit such pattern.

[1] In our prototype, we rely on the bytes corresponding to the ASCII code of "LKP", which stands for LightKone Protocol, in the 14^{th} byte of each message.

We have implemented the Yggdrasil Low Level Library as a fully isolated component for two primary reasons, which are: *i*) it allows the reuse of this library in other contexts other than the Yggdrasil framework, for instance, by having this library as an isolated component one can wrap its implementation to allow its usage in other programming environments such as Erlang; and *ii*) it simplifies the replacement of this library by another implementation (exposing an equivalent interface) that can correctly operate in non linux environments, such as embedded systems or sensors.

Yggdrasil Runtime: This is the core component of our framework. Its main responsabilities are as follows: *i*) it manages the life cycle of all components in the framework, and in particular it manages the execution threads associated with each protocol and application that are loaded; *ii*) it exposes a set of communication mechanisms that can be used by applications and protocols to interact with each other. These mechanisms are, in a nutshell, an interface for protocols/applications to emit and register their interest in specific *events*, which resembles a publish-subscribe interface [39] and an interface for a protocol/application to issue a *request* for another specific protocol or applications, and for a *reply* to be sent back; *iii*) it exposes an API to allow applications to specify the protocols required for its operations and to specify the requirements of the application in terms of the wireless network to be used; and *iv*) it exposes interfaces to interact with the essential support protocols used by the framework, which we detail below.

Essential Support Protocols: Since Yggdrasil adopts an event driven execution model for the design, a common necessity when implementing applications and protocols in Yggdrasil (and in general) is to send/receive messages and manage local timers (either to capture a time-out condition, or to allow the execution of a periodic task). To simplify these tasks, Yggdrasil features two essential support protocols. An API to interact with these protocols is provided by the Yggdrasil Runtime.

- Dispatcher Protocol: This protocol is responsible for managing the direct interactions with the channel abstraction provided by the Yggdrasil Low Level Library. In particular this protocol is responsible for sending and receiving messages through the radio interface. This protocol also features a mechanism that enables a protocol or application to block (and unblock) a given device identified by its radio MAC address. When a MAC address is blocked, all messages that are received from that device are discarded by the Dispatcher. This feature is useful for testing purposes, such as emulating a multi-hop network when all devices are physically collocated or to block nodes that exhibit link flapping behavior [69].
- Timer Protocol: This protocol is responsible for managing the life cycle of all timers used by any protocol or application. The protocol supports timers that are triggered a single time or that are triggered periodically. It features mechanisms for other protocols and applications to create new timers and cancel existing timers. When a timer expires, the protocol is responsible for notifying (resorting to the Yggdrasil Runtime) the appropriate protocol or application as to execute the correspondent handler for that timer.

Our implementation of the Yggdrasil support protocols, are similar to any other user level Yggdrasil protocol, and in fact, their execution is managed by the Yggdrasil Runtime in a similar fashion to any other user protocol. While this can add some overhead compared with providing these functionalities directly as part of the Yggdrasil Runtime, we believe that this is a more sensible design choice, as it empowers developers to write their own variants of these protocols and replace these protocols in concrete applications. We envision this as being a useful feature in particular, for performing debugging (enabling the logging of all messages and timers generated by the framework) as well as to perform experimental assessment of the performance of protocols and applications, by using instrumented variants of these protocols that extract quality metrics.

User Level Protocols: User level protocols can provide any form of functionality required by an application. Protocols in Yggdrasil are identified by a unique global numerical identifier, which is defined when the protocol is registered by an application. When a protocol is initialized, it can receive a set of runtime parameters, that are defined by the application requesting the execution of the protocol. The two properties described above, allow multiple instances of the same distributed protocol to be executed within the context of a Yggdrasil process if different applications require it, or for multiple applications to share a single instance of a given protocol.

Protocol implementation is simple when using Yggdrasil. Protocols are defined by a main control function that is executed by a thread controlled by the Yggdrasil Runtime. This function, usually features a control loop, where the protocol waits for an event to be delivered to it. Upon the reception of an event, the protocol will verify the type of event and execute its handler for that event type. Such handlers can generate new events (e.g, register new timers, send messages, send notifications, issue requests or replies).

Events are delivered to protocols (and applications) through a queue, that is also managed by the Yggdrasil Runtime. This queue features an API that allows a protocol to block (and for its thread to yield the processor) waiting for the next event indefinitely or for a limited amount of time.

For validating the design and implementation of Yggdrasil, we have designed and implemented a set of simple distributed protocols that offer common features that are useful for the design and implementation of distributed applications. In more detail, we have designed and implemented the following protocols:

- Neighbour Discovery Protocols: This is a class of protocols that offer to the local process a local view of the other processes in the system, considering all or a sub-set of the direct neighbours of that process. In this context, we consider a direct neighbour of process a to be another process b such that messages can be directly exchanged from b to a. We have implemented three different protocols for this class: i) a total neighbour discovery protocol, that exposes a list with all processes reachable from the local process; ii) a partial neighbour discovery protocol, that exposes a list with at most K randomly selected processes reachable from the local process; and iii) a dynamic neigh-

bour discovery that behaves in a similar fashion to the total neighbour discovery protocol, except that it resorts to a fault-detection protocol (described below) to infer the arrival and departure (i.e, due to failure or shut-down) of neighbours.

- Fault-Detection Protocols: This class of protocols is responsible for suspecting that processes have failed and for producing notifications to other protocols or applications about these events. Optionally, this class of protocols can also operate as a neighbour discovery protocol. We have implemented two protocols that fall within this class: *i*) a simple failure detection protocol, that resorts to piggyback messages to monitor the continuous activity of neighbouring processes. When a process does not issue a message for a configurable amount of time T, this protocol issues a suspect notification to all interested protocols or applications (through the event notification interface exposed by the Yggdrasil Runtime); and *ii*) a reactive failure detection protocol that behaves similarly to the previously discussed protocol, but that also detects events typically denoted as *link flapping* where, due to poor reachability, a process is continuously suspected and observed as active. In this case, this protocol resorts to the feature of the Dispatcher protocol that allows to blacklist a device, by adding the MAC address of such node to the black list for a configurable amount of time.
- Membership Protocols: This class of protocols provide processes with either complete or partial views of the system membership that are not restricted by the capacity of processes to communicate directly. We have implemented a single global membership protocol that provides each process with a complete view of the system membership. This protocol relies on one of the fault-detection protocols to allow the membership to react to the departure, or failure, of a process.
- Application Level Broadcast Protocols: A common requirement of distributed applications is to disseminate information across all processes of the system. To this end, one usually resorts to an application level broadcast. We have implemented such a protocol based on an eager-push gossip protocol. The protocol offers probabilistic guarantees of delivery, and resorts to random delays in re-transmissions to minimize the risk of message collisions in the wireless medium.

Application(s): In Yggdrasil, applications have the same structure of protocols. Applications also have a queue that allows them to receive events and can also generate events as any protocol. As discussed previously, Yggdrasil allows for multiple applications to execute within the context of a single Yggdrasil process.

In addition to the components described above, we also have implemented multiple test applications for the features discussed above, and simple demo applications that showcase how to use our framework.

The prototype of the framework was also used in a test pilot in the context of an advanced course taught by Professor João Leitão at the Faculdade de Ciências e Tecnologia of the NOVA University of Lisbon. The course, entitled *Algorithms and Distributed Systems*, is an advanced course of the Integrated Master's Program in Computer Science,

and teaches the fundamental aspects on the design, implementation, and correctness of distributed algorithms. The framework was used by a group of two students to develop a global membership protocol and an aggregation protocol on top of the framework.

In fact, the framework described here can easily be used for teaching advanced courses in distributed algorithms, by allowing students to design and experiment with distributed protocols operating in AdHoc Wireless networks, and in particular protocols that resort to MAC level communication, using commodity hardware, such as Raspberry PIs or even student's laptops running linux. We plan to further explore this venue in the future.

(c) Aggregation Protocols

As a way to showcase the benefits of our framework for supporting the development and execution of distributed protocols, and also to pave the way for supporting general-purpose computations in this edge environment, we have developed three different aggregation protocols. All our aggregation protocols operate over numerical data, that we assume is distributed across all nodes in the system[2]. Our algorithms support the following four aggregation operators: maximum, minimum, sum, and average.

All our protocols are reactive, in the sense that the protocol only computes the intended value when an application, or another protocol, explicitly requests the computation. Background aggregation protocols, that continuously maintain a long-running computation to keep up-to-date aggregations of values that continually change in each independent edge device are also interesting, and will be addressed as part of future work.

The three aggregation protocols that we implemented are (the pseudo-code for these aggregation protocols is presented in Appendix A for completeness):

Dissemination Aggregation: This aggregation algorithm operates by having each process perform a one hop broadcast of all contributions (i.e, individual values tagged with the unique identifier of the source process) that the process knows locally. Initially a process simply disseminates its own local value. Upon reception of this message a process adds the received value to a local set of values (that is always initialized with the local process value) and applies the requested aggregation function, computing a partial result. After some time, the process disseminates this set to all its direct neighbours. After initializing the protocol (either by an explicit request of a local protocol or applications, or through the reception of the values from another process) each process will continue to disseminate its own set periodically every T time units (T is a configurable parameter of the protocol). Upon every reception of a set, a process first extracts the contributions that are not present in its own local set. After this, it adds each contribution to its local set and applies the requested aggregation function.

The protocol terminates in the following way. After a process disseminates its local set of known contributions C times (C is another parameter of the protocol) without receiving any message from another process that leads the local known set of contributions to grow, the process considers that there are no other contributions

[2]Such data will usually be acquired from sensors that are in the vicinity of each edge device, where each process executes. For testing these algorithms we have provided each process with a static and distinct numerical input.

to acquire. This leads the algorithm to terminate and the protocol to issue a reply to the protocol or application that requested the aggregation result (evidently, this last part only takes place in the process where the request was originally issued).

Bloom Aggregation: This aggregation algorithm is very similar to the previous one, with some modifications that address a scalability limitation of the Dissemination Aggregation algorithm, that is, for a large number of processes, the size of the set propagated among processes might become excessively large. To avoid this, the Bloom Aggregation algorithm instead propagates among processes the following information: its own contribution, the partial-result obtained through the application of the target aggregation function over the received information, the number of different processes that contributed for that partial-result, and a bloom filter [22] (of configurable size) containing the identifiers of all processes that have contributed to the disseminated partial-result. Furthermore, each process stores the contributions of their direct neighbours.

In this algorithm however, the processing of received messages is somewhat more complex and, in a nutshell, proceeds with the following steps:

1. If the received bloom filter is exactly the same as the locally stored bloom filter (notice that each process begins the execution of the algorithm with a bloom filter containing its own unique identifier), then the process does not modify its state.
2. If the received bloom filter is completely divergent from the locally stored bloom filter (i.e, there is no bit set to one in both bloom filters) then the process merges the two bloom filters (with a binary OR operation) and recomputes its local partial-result based on the number of locally known contributions and previous partial-result and the partial-result and number of contributions in the received message.
3. If the local stored bloom filter does not contain the identifier of the process that sent the message, then the locally stored partial-result and bloom filter are updated to take into account the individual contribution of the process that sent the message.
4. If the received bloom filter does not contain the identifier of the local process or the identifier of the locally stored contributions (i.e, from direct neighbours), the received partial-result and bloom filter are updated to include the missing contributions, present in the local node.
5. In the two previous cases, the local process stores the partial-result (and associated bloom filter) that contains contributions from more processes. In the case of a tie (and in the presence of different partial-results) an heuristic is used to select one of those (currently we pick one at random).

The termination of the algorithm is similar to the one described for the Dissemination Aggregation algorithm above, with the key difference that the local condition for termination depends on the locally stored bloom filter remaining unchanged for N periodic transmission steps.

This algorithm fundamentally trades additional processing power and potentially a larger time until the computation of the average terminates to enable the transmission of smaller messages. This is only useful in deployments with large numbers of processes.

Single Tree Aggregation: This algorithm is an implementation of an algorithm originally proposed in [17] adapted to take advantage of the one hop broadcast primitive.

The algorithm operates as follows. Upon receiving a request for performing an aggregation operation by a protocol or application, this algorithm resorts to a neighbour discovery protocol to obtain the list of all direct neighbouring processes. After this, the process broadcasts a request to execute the target aggregation operation. When a process receives such a message for the first time, it replies to the originator of the message with a YES message, otherwise, if the request had already been received from another process, a NO message is sent back. When a node replies with a YES message it also stores the identifier of process from which it received the aggregation request as being its *parent* and re-transmits that request.

Upon receiving the YES message, a process moves the identifier of the sender of that message from its neighbour list to its *children* list. When a process receives a NO message, it simply removes that process from the neighbour list.

After this initial step, a tree is established across the paths over which a YES message was sent. Then the aggregation step is conducted. In this step, processes that have received no YES messages from their neighbours (that don't have any children) send their individual value. After receiving a message from one of its *children*, a process first computes a partial results, given the reported value of his *children* and the locally stored value, and then proceeds to remove the *children* from its *children* set. Once there are no *children* left in the set, the process reports the partially computed result to its *parent*. This process continues up-stream through the tree, until reaching the root (i.e, the process that started the aggregation). This process can then compute the final result and report it to the protocol or application that requested it.

This protocol however is not tolerant to failures while the aggregation is happening. If a fault detector protocol notifies a process that one of its children is suspected as failed, then two options are available: *i*) we abort the aggregation operation by sending a failure notification up-stream through the tree; or *ii*) the process removes the suspected process from its *children* set and continue executing the algorithm regularly. The last option can lead to the computation of an incorrect result for the aggregation, as the result will disregard not only the contribution of the process that failed, but also of any process that was below in the tree. Our current implementation relies on the later strategy.

The protocols presented above are only used to illustrate how one can leverage Yggdrasil to build effective aggregation protocols. In the future we plan to implement more sophisticated protocols, and also explore how to build effective protocols that can continually run aggregation computations in the background across all nodes.

The experimental comparison on the performance of these and other aggregation protocols for AdHoc network scenarios, will be conducted in the context of Work Package 7 and initial results will be reported in deliverable D7.1.

(d) State of the Art

Designing and implementing distributed systems and protocols is an inherently challenging task, and previous work has addressed these challenges by proposing frameworks and runtime environments that ease the life of developers. As we discuss here however, most of these frameworks target either wired environments or more specifically cloud infrastructures.

In the context of supporting the design and implementation of cloud based distributed algorithms and computations, Apache Hadoop is a well known framework to support distributed computations based on map-reduce [33]. Apache Spark [95] is a framework for supporting computations based on stream processing. Both of these frameworks target cloud or grid infrastructures and operate above the Internet Protocol (IP) not supporting computations to be performed in AdHoc settings.

A recent framework called Ovid [13] was proposed for simplifying the reconfiguration of large-scale distributed applications in the cloud. While this framework is more general-purpose than the previous, that focus in particular patterns of distributed computing, Ovid is tailored towards cloud environments, lacking support for AdHoc Networks. We note however, that some of the solutions proposed by Ovid can be useful in enriching Yggsadril to include mechanisms to simplify self management and reconfiguration of distributed computations and applications.

There are two relevant frameworks that, similarly to Yggdrasil, focus on the composition of distributed protocols to support complex applications. These are Isis [19, 21] and Appia [64]. Both of these frameworks have found success in industry, particularly Isis that was used, for instance, to support applications in the Stock exchange domain. Appia, was at some point used to support the replication solution of MySQL. Both of these systems however focus on supporting the execution of protocols associated with the view synchrony and group communication abstractions, and hence are only suitable for wired settings. In contrast, our framework focus on providing mechanisms and supporting protocols, specifically tailored for AdHoc settings.

(e) Discussion

In this section we presented the design of Yggdrasil and discussed some of the protocols currently implemented in this framework. The current design of Yggdrasil focuses on the fundamental requirements for supporting the execution of distributed protocols and applications in wireless AdHoc networks, which is a particularly challenging domain in the context of edge computation.

We believe that Yggdrasil already offers an interesting setting of features and protocols that enable the use of the framework to conduct real experimentation and evaluation, in commodity hardware, of distributed algorithms for AdHoc environments. This is a significant benefit, since, to the best of our knowledge, most algorithms found in the literature for this domain where only experimentally validated through simulation.

Additionally, the current version of Yggdrasil and accompanying protocols, form an interesting tool-kit for use in pedagogical activities, particularly in advanced courses re-

lated with wireless AdHoc protocols and applications, offering a set of abstractions that allow students to develop and experiment with protocols in this domain in commodity hardware.

Notwithstanding, there is significant space for Yggdrasil to grow and offer additional features. In particular we consider the following venues interesting directions to enrich Yggdrasil:

- Offering support for other runtime execution environments, such as single-process operative systems as the ones used in the GRiSP board and other IoT devices.
- Integration of support for allowing applications and protocols in Yggdrasil to interact with sensor/actuators devices, typically used in sensor networks, as to enable processes running Yggsadril applications to gather information and issue commands over these devices. This implies enriching Yggdrasil with support for Channels operating over the ZigBee [74] and 6LowPan [66] protocols.
- Offering support for self-management of protocols and applications through the integration of distributed monitoring schemes.

3.2 Data Computation at the Edge with the Legion Framework

Legion addresses another edge computation scenario, in particular that of peer-to-peer computations performed in end-user devices. While a lot of work as been conducted in developing support for the construction and deployment of peer-to-peer applications [20, 43, 43, 53–56, 75, 81, 91, 93] Legion addresses a different challenge: transparent support for web applications.

A large number of web applications mediate interactions among users. Examples are plentiful, from collaborative applications, to social networks, and multi-user games. These applications manage a set of shared objects, the application state, and each user reads and writes on a subset of these objects. For example, in a collaborative text editor, users share the document being edited, while in a multi-user game the users access and modify a shared game state. In these cases, user experience is highly tied with how fast interactions among users occur.

These applications are typically implemented using a centralized infrastructure that maintains the shared state and mediates all interactions among users. This approach has several drawbacks. First, servers become a scalability bottleneck, as all interactions have to be managed by them. The work performed by servers has polynomial growth with the number of clients, as not only there are more clients producing contributions but also each contribution must be disseminated to a larger number of clients. Second, when servers become unavailable, clients become unable to interact, and in many cases, they cannot even access the application. Finally, the latency of interaction among nearby users is unnecessarily high since operations are always routed through servers. This might not be noticeable for applications with low interaction rates such as social networks. However, user experience in games and collaborative applications relies on interactive response times below 50ms [48].

One alternative to overcome these drawbacks is to leverage on a edge computing approach making the system less dependent on the centralized infrastructure. Besides avoiding the scalability bottleneck and availability issues of typical web applications,

leveraging on edge computing can also bring lower latency of interactions among clients. Additionally, it has the extra benefit of lowering the cost of dedicated centralized infrastructure.

At its core, Legion leverages on recent advances in browser technology, such as the new interfaces and functionalities offered by HTML5 and the capability of browsers of extablishing direct communciation channels through WebRTC [8], to enable the clients of web applications to interact directly. To support these interactions, each client maintains a local data store with replicas of a subset of the shared application objects. These data objects are replicated under an eventual consistency model, allowing each client to locally modify its replica without coordination. Updates are propagated asynchronously to other replicas by the Legion framework. To guarantee that all replicas converge to the same state despite concurrent updates, Legion relies on Conflict-free Replicated Data Types (CRDTs) [78].

This section is organized as follows. § 3.2 (a) introduces the system model and assumptions underlying the Legion design. An overview of the Legion architecture is presented in § 3.2 (b) and the design of each component is detailed in § 3.2 (c). § 3.2 (d) presents two demonstrators that we have constructed to showcase publicly the framework. We discuss some of the public dissemination activities using Legion in § 3.2 (e). Finally, § 3.2 (f) overviews the state of the art and § 3.2 (g) concludes this section providing some interesting directions for future work.

(a) System Model

Legion is a framework for data sharing and communication among web clients. It allows programmers to design web applications where clients access a set of shared objects replicated at the client machines. Web clients can synchronize local replicas directly with each other. For ensuring durability of the application data as well as to assist in other relevant aspects of the systems operation (discussed further ahead), Legion resorts to a set of centralized services. We designed Legion so that different Internet services (or a combination of Internet services and Legion's own support servers) can be employed. These services are accessed uniformly by Legion through a set of *adapters* with well defined interfaces.

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

Legion current design and prototype assumes that applications have tens to few hundreds of users accessing a sharing the same state (i.e, the same data container).

(b) Overview and Architecture

By replicating objects in web clients and synchronizing in a peer-to-peer fashion, Legion reduces both dependency and load on the centralized component (as the centralized component is no longer responsible for propagating updates to all clients), and minimizes latency to propagate updates (as they are distributed directly among clients). Furthermore, it allows clients (already running) to continue interacting when connectivity to servers is lost (for instance, due to a transitive network failure such as a partition).

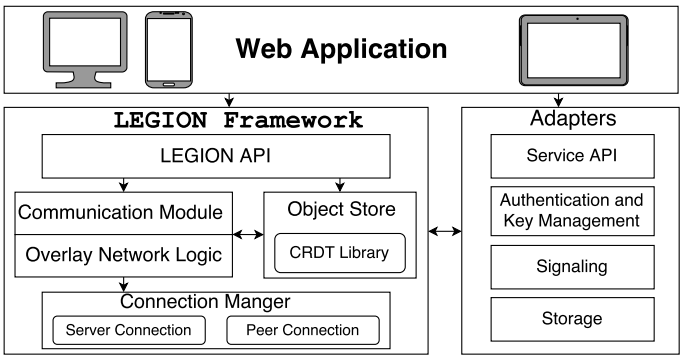


Figure 3.2: Legion Architecture Overview

Figure 3.2 illustrates the client-side architecture of Legion with the main components and their dependencies/interactions. We now discuss the goals of these components and in the following section discuss the design and implementation that we have pursued when developing Legion.

Legion API: This layer exposes the API through which applications interact with our framework.

Communication Module: The communication module exposes two secure communication primitives: point-to-point and point-to-multipoint. Although these primitives are available to the application, we expect applications to mostly interact using shared objects stored in the object store.

Object Store: This module maintains replicas of objects shared among clients, which are grouped in *containers* of related objects. These objects are encoded as CRDTs [78] from a pre-defined (and extensible) library including lists, maps, strings, among others. Web clients use the communication module to propagate and receive updates to keep replicas up-to-date.

Overlay Network Logic: This module establishes a logical network among clients that replicate some (shared) container. This network defines a topology that restricts interactions among clients. Therefore, only overlay neighbours maintain (direct) WebRTC connections among them and exchange information directly.

Connection Manager: This module manages connections established by a client. To support direct interactions, clients maintain a set of WebRTC connections among them. (Some) Clients also maintain connections to the central component, as discussed further ahead.

Legion uses two additional components that reside outside of the client domain:

- one or more *centralized infrastructures* accessed through adapters for: (i) user authentication and key management (*authentication and key management* adapter); (ii) durability of the application state and support for interaction with legacy clients (*storage* adapter); (iii) exposing an API similar to the server API, thus simplifying porting applications to our system (*service API* adapter); (iv) assisting clients to initially join the system (*signaling* adapter);

- a set of STUN [62] servers, used to circumvent firewalls and NAT boxes when establishing connections among clients.

While we have also developed particular adapters for the Google Drive Real Time API (GDriveRT) and for a simple Node.js [50] server implemented by us, in this document we do not detail the design and implementation of adaptors for GDriveRT. The interested reader is redirected to [89] for additional details (the full publication is included in this document on § 9 for convenience).

(c) Legion Design

We now detail the main design and implementation decisions for each of the main components in the Legion framework.

Communication Module The communication module exposes an interface with point-to-point and multicast primitives, allowing a client to send a message to another client or to a group of clients. In Legion, each container has an associated multicast group that clients join when they start replicating an object from the container. Updates to objects in a container are propagated to all clients replicating that container.

Messages are propagated through the overlay network(s) provided by the *Overlay Network Logic* module. The multicast primitive is implemented using a push-gossip protocol (similar to the one presented in [56]).

Messages exchanged among clients are protected using a symmetric cryptographic algorithm, using a key (that is associated with each container) which is shared among all clients and obtained through the centralized component. Clients need to authenticate towards the centralized component to obtain this key, ensuring that only authorized (and authenticated) clients are able to observe and manipulate the objects of a container.

Overlay Network Logic Legion maintains an independent overlay for each container, defining the communication patterns among clients (*i.e.,* which clients communicate directly). The overlay is used to support the multicast group associated with that container.

Our overlay design is inspired by HyParView [56]. It has a random topology composed by symmetric links. Each client maintains a set of $\mathtt{K}$ neighbours (where $\mathtt{K}$ is a system parameter with values typically below 10 for medium scale systems). Overlay links only change in reaction to external events (clients joining or leaving/failing).

In contrast to HyParView, we have designed our overlay to promote low latency links. As such, each client connects to $\mathtt{K}$ peers, with $\mathtt{K} = \mathtt{K}_n + \mathtt{K}_d$, where $\mathtt{K}_n$ denotes the number of *nearby* neighbours and $\mathtt{K}_d$ denotes the number of *distant* neighbours.

As shown by previous research [54], each client must maintain a small number of distant neighbours when biasing a random overlay topology to ensure global overlay connectivity and yield better dissemination latency while retaining the robustness of gossip-based broadcast mechanisms.

This requires clients to classify potential neighbours as being either nearby or distant. A common mechanism to determine whether a potential neighbor is nearby or distant is to measure the round-trip-time (RTT) to that node [54]. However, in Legion, since clients are typically running in browsers, it is impossible to efficiently measure round

trip times between them, since a full WebRTC connection would have to be setup, which has non-negligible overhead due to the associated signalling protocol.

To circumvent this issue we rely on the following strategy that avoids clients to perform active measurements of RTT to other nodes. When a client starts, it measures its RTT to a set of W well-known web servers through the use of an HTTP HEAD request (the web servers employed in this context are given as a configuration parameter of the deployment). The obtained values are then encoded in an ordered tuple which is appended to the identifier of each client. These tuples are then used as coordinates in a virtual Cartesian space of W dimensions. This enables each client to compute a distance function between itself and any other client c given only the identifier of c.

Connection Manager This module manages all communication channels used by Legion, namely *server connections* to the centralized infrastructure, and *peer connections* to other clients. We now briefly discuss the management of these connections.

Server Connections: A server connection offers a way for Legion clients to interact with the centralized infrastructure. We have defined an abstract connection that must be instantiated by the adapters that provide access to the centralized services. Independently of the employed centralized component, server connections are only kept open by a small fraction of clients.

Peer Connections: A peer connection implements a direct WebRTC connection between two clients [3]. To create these connections, clients have to be able to exchange – out of band – some initial information concerning the type of connection that each end-point aims to establish and their capacity to do so, which also includes information necessary to circumvent firewalls or NAT boxes using STUN/TURN servers. This initial exchange is known, in the context of WebRTC, as *signalling*.

Legion uses the centralized infrastructure for supporting the execution of the signalling protocol between a client joining the system and its initial overlay neighbours (that have to be active clients *i.e.,* with active server connections). After a client establishes its initial peer connections, it starts to use its overlay neighbours to find new peers. In this case, the signalling protocol required to establish these new peer connections is executed through the overlay network directly.

Object Store The object store maintains local replicas of shared objects, with related objects grouped in containers. Client applications interact by modifying these shared objects. Legion offers an API that enables an application to create and access objects.

CRDT Library Legion provides an extensible library of data types, which are internally encoded as CRDTs [78]. Objects are exposed to the application through (transparent) object handlers that hide the internal CRDT representation.

The CRDT library supports the following data types: Counters, Strings, Lists, Sets, and Maps. Our library uses Δ-based CRDTs [90], which are very flexible, allowing replicas to synchronize by using deltas with the effects of one or more operations, or the full state.

Causal Propagation: This module uses the multicast primitive of the *Communication module* to propagate and receive deltas that encode modifications to the state of local

[3] Our experiments have shown that WebRTC connections can be established even among mobile devices using 3G/4G connectivity when devices use the same carrier.

replicas in a way that respects causal order (of operations encoded in these deltas). To achieve this, we use the following approach.

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

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

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

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

Security Mechanisms Allowing clients to replicate and synchronize among them a subset of the application state offers the possibility to improve latency and lower the load on central components. However, it also leads to concerns from the perspective of security, in particular regarding data privacy and integrity. In more detail, *privacy* might be compromised by allowing unauthorized users to circumvent the central system component to obtain copies of data objects from other clients; additionally, *integrity* can be compromised by having unauthorized users manipulate application state by propagating their operations to authorized clients.

We assume that an access control list is associated with each data container, and that clients either have full access to a container (being allowed to read and modify all data objects in the container) or no access at all. While more fine-grained access-control policies could easily be established, we find that this discussion is orthogonal to main design challenges of Legion. We also assume that the centralized infrastructure is trusted and provides an authentication mechanism that ensures that only authorized clients can observe and modify data in each container. Finally, we do not address situations where authorized clients perform malicious actions.

Considering these assumptions, Legion resorts to a simple but effective mechanism that operates as follows. The centralized infrastructure generates and maintains, for each container C, a persistent symmetric key K_C within that same container. Due to the authentication mechanism of the centralized infrastructure, only clients with access to a container C can obtain K_C. Every Legion client has to access the infrastructure upon bootstrap, which is required to exchange control information required to establish direct

connections to other clients. During this process, clients also obtain the key K_C for the accessed container C.

K_C is used by all Legion clients to encrypt the contents of all messages exchanged directly among clients for container C (with the exception of the version number of K_C). This ensures that only clients that have access to the corresponding container (and have authenticated themselves towards the centralized component) can observe the contents and operations issued over that container, addressing data privacy related challenges.

Whenever the access control list of a container is modified to remove some user, the associated symmetric key is invalidated, and a new key for that container is generated by the centralized infrastructure (we associate an increasing version number to each key associated with a given container).

To enable clients to detect when the key is updated in a timely fashion, the centralized component periodically generates a cryptographically signed message (using the asymmetric keys associated with the certificate of the server, used to support SSL connections) containing the current version of the key, and a nonce. This message is sent by the server to active peers, that disseminate the message (without being encrypted) throughout the overlay network.

If a client receives a message encrypted with a different key from the one it knows, either the client or its peer have an old key. When the client has an old key (with a version number smaller than the version number of the key used to encrypt the message), the client contacts the centralized infrastructure to obtain the new key. Otherwise, the issuer of the message has an old key and the client discards the received message and notifies the peer that it is using an old key. This will lead the sender of the message to connect to the central infrastructure (going again through authentication) to update the key before re-transmitting the message.

Note that clients that have lost their rights to access a container are unable to obtain the new key and hence, unable to modify the state of the application directly on the centralized component, send valid updates to their peers, or decrypt new updates.

(d) Demonstrators

Multi-user Pacman: We adapted a JavaScript version of the popular arcade game Pacman [4] to operate under the GDriveRT API with a multi-player mode. We also added support for multiple passive observers that can watch a game in real time. In our adaptation up to 5 players can play at the same time, one player controlling Pacman (the hero) and the remaining controlling each of the four Ghosts (enemies).

The Pacman client is responsible for computing, and updating the adequate data structures, with the *official* position of each entity. Clients that control Ghosts only manipulate the information regarding the direction in which they are moving. If no player controls a Ghost, its direction is determined by the the original game's AI, running in the client controlling Pacman.

In this game, we employed the following data types provided by the GDriveRT API: (i) a map with 5 entries, one for Pacman and the remaining for each Ghost, where each entry contains the identifier (ID) of the player controlling the character (each user generates its own random ID); (ii) a list of events, that is used as a log for relevant game events, which include players joining/leaving the game, a Ghost being eaten, Pacman being captured, etc. (iii) a list representing the game map, used to maintain a synchronized

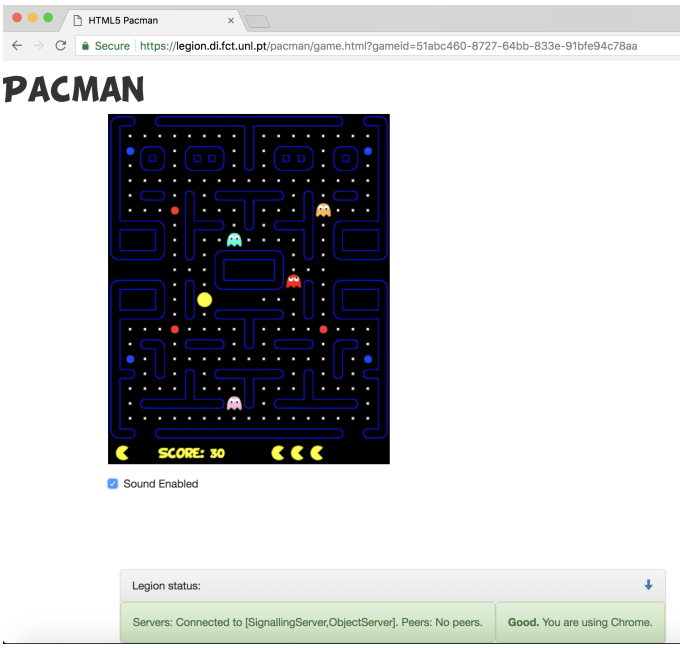


Figure 3.3: Legion Pacman Running in the Browser

view of the map between all players. This list is modified, for instance, whenever a *pill* is eaten by Pacman; (iv) a map with 2 entries, one representing the width and the other the height of the map. This information is used to interpret the list that is used to encode the map; (v) a map with 2 entries, one used to represent the state of the game (paused, playing, finished) and the other used to store the previous state (used to find out which state to restore to when taking the game out of pause); finally, (vi) 5 maps, one for each playable character, with the information about each of these entities, for maintaining a synchronized view of their positions (this is only altered when the corresponding entity changes direction, not at every step), directions, and if a ghost is in a vulnerable state.

After concluding the implementation of the game to use the GDriveRT API, modifying the game to leverage on Legion (using our GDriveRT adapters) required to change only two lines of code. Figure 3.3 illustrates this demonstrator.

Figure 3.4: Legion Shooter Running in the Browser

Legion Shooter To showcase the flexibility of the Legion framework, we also developed a second game that does not resort to the Object Store component of Legion, and instead operates by having clients disseminating messages directly among them. This is a very simple Shooter, where each player controls a small *square* in a battlefield. Each player can shoot bullets in any of the 4 main directions (up, down, left, and right). The game shows associated with the *square* of each player the username of the player and its score (how many times he destroyed an enemy and how many times he destroyed an enemy since spawning).

The game is modelled though a set of native javascript objects that are loaded together with the web page: (i) a map that is represented by a matrix of numbers that encode what is in each position of the map (free space, a wall, or a position where a player should not be spawned) and two integers that define the height and length of the map; (ii) a string with the username selected by the player; (iii) two integers representing the position of the player; (iv) a map with the player id, username, and current position of each player in the game; and (v) a list containing the list of bullets in the field (which includes information for the player that shot that bullet, its current position, and its speed).

The web client is responsible for computing the effects of fog of war (hiding from the player the locations of the map for which he does not have a direct line of sight), computing the positions of bullets, updating the score of the local player, and gathering the input from the user. Figure 3.4 shows the shooter application, from the perspective of a non-player user (left) that observes the whole game and from the perspective of a player that is conditioned by the fog of war.

The distributed logic of the game is achieved by having each client disseminate a message whenever: (i) a player spawns in the map, to notify all other clients of the new location of the client; (ii) a player modify its movement, to update the new location and direction of movement for the local player; (iii) a player shoots, to create the new bullet with the associated relevant information; (iv) a player kills an enemy, to notify the killed player of the event, and update the score of both players.

While the game was extremely easy to implement, it has been a success in public demonstrations of the framework, as we detail in the following section.

(e) Dissemination Activities

We have used Legion as a demonstrator of the LightKone project results and also of the research activities conducted by NOVA in two public events. We now briefly report on these dissemination activities.

ExpoFCT (April 2017): The ExpoFCT is a public event hosted at Faculdade de Ciências e Tecnologia of the NOVA University of Lisbon where hundred of high school kids and their professors (as well as some general public) visit the school and each department offers a set of activities that showcase the research and development activities conducted in the department. In the 2017 edition, the Legion framework was one of the activities put forward by the department, where a room with 15 desktop computers was made available for visitors to experiment with the Legion demonstrators described above.

During this day, we had more than 200 users trying our demonstrators, most of them were high school students. We have explained to them what made that new,

and the benefits that could be extracted from the use of the Legion framework. Overall the reaction of the users were highly positive. They were very happy playing the games (particularly the shooter game, that allowed all player in the room to play against each other) and were also impressed with the new technology, and the fact that this could easily be used from their homes.

Science 2017 National Meeting (July 2017): The Science 2017 National meeting, is a national event organized by the Portuguese government and directed at the general public to showcase highlights of the research results achieved by Portuguese research centers and Universities. In 2017, Legion was one of the demo highlights put forward by the NOVA LINCS laboratory in one the days, where the usefulness of the framework was shown through the use of mobile devices (smart phones and tablets) using the demonstrators reported previously. The underlying technology was explained to the attendees that interacted with the demos.

The event allowed to expose these research results many dozens of people. Overall, and similar to the other events that leveraged on our demonstrators to showcase the benefits of Legion, attendees where impressed that this new technology could immediately be used by them by simply accessing a web page that was leveraging on the Legion framework.

The web page of the event can be found in http://encontrociencia.pt/home/.

Closing Session of the CodeMove PT (December 2017): The CodeMove PT is a national movement in Portugal, sponsored by the Portuguese Government (through the Ministry for Science, Technology, and Higher Education, the Ministry for Education, and the Ministry for Work, Solidarity, and Social Security and the Ministry for Economy) that hosts a final public event during the weekend that occurs for a whole day. Since NOVA is also an active partner of this movement, we were invited to this event, and one of the presentations that we had for the general public was the Legion framework through the use of the same demonstrators, in four computers that were made available at the event location.

During the event, we had close to 100 users trying or demonstrators. Again the reactions were extremely positive, and most of the attendees, when we explained the underlying technology, were impressed by the fact that this was new technology that could be readily available at their homes.

The web page of the event can be found in https://www.codemove.pt.

Overall, these dissemination activities were useful to both promote the results of the project to the general public, and also to reinforce the idea that simple demonstrators and the fact that new edge technology can easily be brought to the everyday life of the general public, makes the results much more impactful.

(f) State of the Art

The Legion framework is a contribution that touches on multiple fields of research of computer systems. In particular we identify four main relevant fields, and briefly discuss how existing solutions in each of these fields is related with our contribution.

Internet services: Internet services often run in cloud infrastructures composed by multiple data centers, and rely on a geo-replicated storage system [11, 29, 31, 34, 59] to store application data. Some of these storage systems provide variants of weak consistency, such as eventual consistency [34] and causal consistency [11, 59], where different clients can update different replicas concurrently and without coordination. Similar to Google Drive Realtime, Legion adopts an eventual consistency model where updates to each object are applied in causal order.

Other storage systems adopt stronger consistency models, such as parallel snapshot isolation [80] and linearizability [31], where concurrent (conflicting) updates are not allowed without some form of coordination. Coordination among replicas for executing each update is prohibitively expensive for high throughput and large numbers of clients (manipulating the same set of data objects).

Replication at the clients: While many web applications are stateless, fetching data from servers whenever necessary, a number of applications cache data on the client for providing fast response times and support for disconnected operation. For example, Google Docs and Google Maps can be used in offline mode; Facebook also supports offline feed access [40].

Several systems that replicate data in client machines have been proposed in the past. In the context of mobile computing [84], systems such as Coda [51] and Rover [49] support disconnected operation relying on weak consistency models. Parse [5], SwiftCloud [96] and Simba [72] are recent systems that allow applications to access and modify data during periods of disconnection. While Parse provides only an eventual consistency model, SwiftCloud additionally supports highly available transactions [14] and enforces causality. Simba allows applications to select the level of observed consistency: eventual, causal, or serializability. In contrast to these systems, our works allows clients to synchronize directly with each other, thus reducing the latency of update propagation and allowing collaboration when disconnected from servers.

Bayou [83] and Cimbiosys [73] are systems where clients hold data replicas and that exploit decentralized synchronization strategies (either among clients [73] or servers [83]). Although our work shares some of the goals and design decisions with these systems, we focus on the integration with existing Internet services. This poses new challenges regarding the techniques that can be used to manage replicated data and the interaction with legacy clients, namely because most of these services can only act as storage layers (i.e., they do not support performing arbitrary computations).

Collaborative applications: Several applications and frameworks support collaboration across the Internet by maintaining replicas of shared data in client machines. Etherpad [38] allows clients to collaboratively edit documents. ShareJS [44] and Google Drive Realtime [45] are generic frameworks that manage data sharing among multiple clients. All these systems use a centralized infrastructure to mediate interactions among clients and rely on operational transformation for guaranteeing eventual convergence of replicas [70, 82]. In contrast, our work relies on CRDTs [78] for guaranteeing eventual convergence while allowing clients to synchronize directly among them. Collab [27] uses browser plugins to allow clients in the same area network to replicate objects using peer-to-peer. Our work uses standard techniques for supporting collaboration over the Internet,

requiring no installation by the end user, and allowing interaction with existing Internet services.

Peer-to-Peer systems: Extensive research on decentralized unstructured overlay networks [43, 56, 93] and gossip-based multicast protocols [20, 26, 56] have been produced in the past. Legion leverages and adapts some of the approaches proposed to implement its peer-to-peer model of direct communication among clients. Although our design for supporting peer-to-peer communication among clients builds on HyParView overlay network [56], it differs from this system in the way it promotes low latency links among clients and leverages the centralized infrastructure for handling faults efficiently.

(g) Discussion

In this section we have presented the design and implementation of Legion, a framework to support the design of user centered web applications, that leverage on edge computing by allowing web clients to directly interact with each other, through the replication of relevant fractions of the application state. Legion focus on particular edge setting of peer-to-peer systems, where applications have their logic partially running in end users devices.

Legion paves the way for a new generation of edge-enable web applications which are increasingly relevant in our society and economy. It also allows small and medium companies to be more competitive in this environment, by minimizing the investment required on centralized (i.e, cloud) infrastructures, while enabling applications that become suddenly popular to gather additional resources from the edge of the system in an organic fashion.

We plan to extend Legion in two fundamental aspects:

- Develop new adapters, that simplify the task of designing edge-enabled distributed applications which have their components executing in other edge scenarios.
- Extend the Legion architecture and design principles to take better advantage of mobile edge computing scenarios, a market that is continually growing now-a-days.

3.3 Data Computation on Lasp

Lasp is a framework that empowers programmers to perform computations over CRDTs in a synchronization free way. Lasp is tailored for running close to the center of edge systems, in either data centers or fog computing environments (such as regional data centers or clusters closer to the edge of the system).

At its core, Lasp provides all the support for expressing (distributed) computations over CRDTs, where CRDTs are replicated across large numbers of nodes. Lasp propagates operations executed over CRDTs and hence, computations in Lasp are themselves replicated across all nodes in a deployment. To achieve this, the Lasp framework combines a set of abstractions, including a programming API, a library of CRDTs, membership and communication abstractions, and a storage component.

Lasp original design was motivated to support mobile applications, particularly mobile applications that operate over fractions of the applications state, encoded as CRDTs

in a disconnected fashion. When these mobile applications can contact the Lasp infrastructure running in data centers, these operations are propagated and synchronized with the infrastructure, leading to the progress of computations and the update of views over CRDTs being maintained and executed by Lasp dynamically.

This sections is organized as follows. § 3.3 (a) describes the system model and provides a brief overview of Lasp. § 3.3 (b) briefly discusses the Lasp architecture. The design of the main components of the Lasp architecture are discussed in § 3.3 (c). § 3.3 (d) explains the role of Lasp in supporting edge computations and § 3.3 (e) briefly mentions the use of Lasp outside of academia and beyond the scope of the Lightkone project. § 3.3 (f) discusses the state of art and finally, § 3.3 (g) provides a final discussion identifying relevant future directions for the evolution of Lasp.

The scalability of Lasp in a practical setting has been reported in [63]. For the convenience of the reader we reproduce the full text of this publication in Appendix.

(a) System Model

Lasp is a framework that combines a distributed programming model, replicated CRDT data store, and distribution layer for Erlang. It's presented as a library, and available for use in new or existing Erlang and Elixir applications. Developers can therefore, leverage Lasp to augment portions of their distributed applications, rather than building their entire applications in Lasp.

Lasp makes no assumptions on the underlying infrastructure. While a default deployment of Lasp assumes that nodes are connected via Distributed Erlang, Lasp provides an alternative distribution layer, named Partisan, that can be used to support the operation of Lasp across different Distributed Erlang clusters. This allows these clusters to grow to larger sizes, or adopt different topologies, than the ones provided by normal Distributed Erlang clusters (i.e, fully connected topologies).

Furthermore, Lasp makes no assumptions on the underlying communications framework. Lasp provides an implementation of an anti-entropy protocol that ensures eventual delivery of all messages in a Lasp system, providing more guarantees to applications operating on top of a variety of different network protocols, such as TCP, UDP, and other frameworks such as publish/subscribe.

Following from this, Lasp assumes eventual consistency at it's core, ensuring the correctness and convergence of computations regardless of update order or visibility. This is ensured by the programming model offered by Lasp, which prohibits expressing computations that would violate this property.

However, Lasp requires a key-value store for the storage of application state. Since Lasp assumes eventual consistency, any storage engine can be plugged in. As such, Lasp automatically takes care of data synchronization and ensures all updates are propagated to all nodes belonging to the system.

(b) Architecture

The Lasp architecture is composed by the following main components:

- **Lasp API:** Lasp API provides the top-level API for adding and removing CRDTs from the scope of an application. It also provides basic data transformations and ag-

gregation operations using CRDTs within the application, allowing to create views over CRDTs.

- **Partisan:** Partisan provides the communication layer used by Lasp. Partisan provides a topology-agnostic programming model for cluster maintenance and point-to-point communication between nodes in the application.
- **Lasp KV:** Lasp KV provides the underlying key-value store inside Lasp which is used for efficient synchronization of data objects (CRDTs) used in a Lasp application.
- **Sprinter:** Sprinter is the deployment system for Lasp applications that eases the task of application deployment across a variety of cloud providers (e.g, Google Cloud Platform, Apache Mesos, and Docker Compose).
- **Types:** Types is a library of CRDTs used within the Lasp KV storage engine.

(c) Lasp Design

We now detail the design and implementation of each of the architectural components of a Lasp system.

Lasp API: Lasp API exposes the programming model for Conflict-Free Replicated Data Types offered by the framework. Lasp allows developers to write application that operate and manipulate CRDT objects, namely by exposing views that aggregate the state of multiple CRDTs. As the source objects of the computation are CRDTs, the functions are either monotone or homomorphisms, and the output objects are CRDTs. This ensures two important properties: (i) compositionality, where the outputs of computations can be used as the inputs for other computations; and (ii) convergence, in that, regardless of message ordering or message duplication, results converge to the same value across all nodes that are executing the (distributed) computation.

Developers using Lasp rely on the provided API to create and update CRDT objects in their applications, which are stored in the underlying key-value store (Lasp KV) and fully replicated across all nodes within the system. Computations are specified by using a set of combinators, such as `map`, `filter`, and `fold` to derive new CRDTs, which are also stored in the key-value store.

Partisan: Partisan is a topology-agnostic programming model and distribution layer for Erlang that is meant to be used to either (i) interconnect independent Distributed Erlang clusters; or (ii) as a full replacement for Distributed Erlang.

Partisan provides a unified API over all of it's supported topologies, which are: (i) full mesh; (ii) client-server; (iii) peer-to-peer; (iv) static; and (v) publish-subscribe. The API provides functions for asynchronous message delivery and cluster membership operations such as joining and removing nodes from the cluster.

Developers using Partisan can implement their own topologies as well, by writing a module that implements the Partisan peer service behaviour, similar to an interface in the Java programming language. Lasp uses Partisan to support node-to-node communication across a cluster of nodes.

Lasp KV: Lasp KV is the underlying storage engine used by Lasp to store CRDT state and propagate it to other nodes in the system. Lasp KV supports two strategies of CRDT dissemination: state-based and delta-based, both configurable at runtime. As Lasp KV is a fully replicated data store, state is asynchronously propagated to all other nodes across the system, using the Partisan distribution layer.

Lasp KV also provides a small layer on top of an existing data store, allowing any existing data store to be used. Hence, Lasp natively supports Redis, Riak, Erlang Term Storage (ETS) and Durable Erlang Term Storage (DETS).

Moreover, Lasp KV interacts with a Lasp system by recomputing views when the source objects that contribute to those views change. This behaviour can be specified through the Lasp API. By default, Lasp KV will fully replicate both the source objects and the derived objects, but this can be configured in the application code, as well.

Lasp KV can, and has, also been used as a standalone storage system, where existing applications use only the get/put API of Lasp KV and its background node synchronization functionality.

Sprinter: Sprinter is a deployment system for Lasp applications that facilitates the operation of large-scale Lasp clusters on Apache Mesos (deployed on any of the existing cloud providers), Kubernetes (deployed locally or via Google Cloud Platform, or Microsoft's Azure), and Docker's Compose. Sprinter only works with Partisan-enabled systems, because it is closely tied to Partisan's membership and clustering API.

Types: Types is an Erlang library of CRDTs that is used by the Lasp components. This library contains two sets of implementations: one for state-based CRDTs (extended with delta-mutator support) and pure-operation based CRDTs. Currently, Lasp only supports the use of state-based and delta-based CRDTs.

(d) Edge Computing over Lasp

By assuming eventual consistency, computations are safe under disconnection and reconnection, message duplication, and message re-ordering. Therefore, Lasp is well suited to an environment where very few guarantees can be made. Lasp was also designed with one specific edge case in mind, providing support for mobile computing where clients replicate state and operate locally, with asynchronous propagation of state changes.

Every component of Lasp can be configured at runtime and was designed to be pluggable. Due to this, Lasp can likewise be used to run experiments where applications can be tested by alternating between different topologies, data structures, and storage engines at runtime, allowing analysis over which combinations yields the best performance under different application scenarios.

(e) Impact and Dissemination Activities

Both Lasp and Partisan have adoption in industry and are available as open-source on GitHub at: https://github.com/lasp-lang

Industry adoption of Lasp has been focused on the use of Lasp as a replicated data storage system tailored for CRDT objects, and also of the Partisan module, which is leveraged to support point-to-point communication in clusters running Erlang software.

(f) State of the Art

Lasp is a fault-tolerant dataflow system. Existing systems such as Apache Hadoop, Apache Spark [95], Apache Flink, Apache Storm, and Google Dataflow are all distributed dataflow systems, however, with different fault-tolerance properties. Designs like Spark provide fault-tolerance through tracking the lineage of computations, and use backup replicas to replace, or recompute, missing or lost information when failures occur. In Spark, computations occur across a series of nodes, and rely on a scheduling strategy for optimal completion of a query with minimal data reorganization. However, most of these systems operate on immutable data. Naiad [67] is a dataflow system that supports fixed-point computations, but is not fault-tolerant, therefore requiring computations be restarted in the event of a failure. Naiad is also designed for immutable data.

Lasp's execution model is closer to materialized view maintenance, where views are constantly refreshed as the source data changes. However, Lasp differs from traditional materialized view maintenance in modern database systems because views are CRDTs: therefore views are mergeable, and views have an ordering relation defined on them which is compatible with their semi-lattice order.

(g) Discussion

The Lasp framework offers support for performing computations in a distributed way over CRDTs without requiring explicit synchronization among replicas to enable progress. This provides an important step on how to support general purpose computations in complex edge-enabled applications. While Lasp core is tailored for running in the core of the system and smaller data centers it can support edge applications running in user devices, such as mobile phones, that operate, in a disconnected way, over fractions of the application state.

Moving forward, we plan to evolve Lasp, and in particular to enrich it with multiple features, that will allow its use in more edge scenarios. In particular, some of the interesting research and development venues for Lasp are:

- **Partial replication:** Lasp currently assumes full replication. However, when considering large number of devices and large amounts of data being produced by edge enabled applications, full replication can easily become a limiting factor for scalability. To address this, we plan to explore how to adapt the computation model of Lasp to take into account partial replication.

- **Causality:** Causality has been shown to simplify the management of application invariants, particularly when there are dependencies among operations that manipulate different fractions of the state of an application. Currently the communication abstractions and replications schemes provided by Lasp do not provide causality. How to efficiently provide this additional guarantee without compromising the computational model of Lasp is going to be addressed in future work.

- **Transactions:** While causality can assist in protecting some application invariants, others require the execution of sets of operations over the application state in an atomic way. Transactions are the common abstraction to enforce this guarantee. Enriching the interfaces provided by Lasp to allow the operation over multiple

CRDTs in the context of transactions can, therefore, be useful to many edge enabled applications.

- **Expressivity:** Finally, the Lasp API that exists today is very simplistic. The current API only supports very simple transformations and aggregation operators. Exposing a richer interface that allows to execute more complex computations is, thus a necessity. Additionally, the API will have to be enhanced to expose abstractions for programmers that capture complex aspects of distributed computations such as causality and transactions.

3.4 GRiSP platform and hardware

Traditional Internet of Things (IoT) development is usually done with custom hardware. Sometimes development uses off-the-shelf components soldered onto custom boards. However, custom hardware makes prototyping and changes in later iterations more expensive. Additionally, errors present in the hardware are also more costly to address if the hardware has to be re-designed and re-built.

Embedded software is often developed using low-level languages. Such languages are closer to the hardware and can be more powerful for hardware access. They also make development more time consuming and error prone. Furthermore, software development in low-level languages is less productive than higher level languages[88].

We have developed a platform called GRiSP to work around these limitations. The goal is to support rapid prototyping and development of IoT edge applications. It consists of a reference base hardware board and an accompanying software. The hardware has connectors for many common interfaces, while the software stack lets you develop applications using a high-level language: Erlang. Erlang is a highly productive functional language, that is also used in other artifacts produced by the LightKone consortium, namely the AntidoteDB (see D6.1). It comes with a soft real-time virtual machine suited for networked edge devices.

GRiSP addresses two concrete edge scenarios, the IoT scenario which is the main focus of GRiSP, while potentially being also useful for application in sensor networks scenarios, particularly for hosting sensors and potentially actuators that can interact with the environment. The development and dissemination of GRiSP is being conducted by Peer Stritzinger GMBH, one of the members of the Lightkone consortium.

(a) Hardware Design

Part of the GRiSP project is a reference platform hardware board that we call GRiSP base. It is a USB powered micro computer.

CPU and Memory The board has a 32-bit System on a Chip (SoC) 300 Mhz ARM Cortex M7 central processing unit (CPU). It has a single and double precision floating point unit (FPU). It also comes with on-board digital signal processing (DSP) extensions. The CPU itself has 384 KiB static random-access memory. This we have enhanced with 64 MiB synchronous dynamic random-access memory (SDRAM).

Storage The board has storage in the form of 2048 KiB flash memory used for the boot loader. A Secure Digital (SD) card slot provides extensible storage for applications.

We have also added 2 KiB electrically erasable programmable read-only memory (EEPROM). Developers can store configuration and other data here which will survive power loss.

Pluggable Hardware The board has several different connectors for pluggable hardware. This makes it easy to prototype IoT edge devices.

- **GPIO:** Two 6-pin General Purpose Input/Output (GPIO) ports with configurable pins.
- **UART:** One 6-pin Universal Asynchronous Receiver/Transmitter (UART) port. UART provides asynchronous serial communication. It has generic bit streams with configurable data format and transmission speed.
- **SPI:** Two Serial Peripheral Interface (SPI) ports, one 6-pin and one 12-pin. SPI provides a synchronous serial communication bus which many chips support. It is fast with speeds up to 20 Mbit/s.
- **I_2C:** One Inter-Integrated Circuit (I_2C) bus connector. I_2C is a primary/secondary protocol where one primary controls secondary devices. Primary and secondaries on the bus communicates with addressable data packets. The protocol is slow with speeds up to 0.4 Mbit/s. It is also unreliable over longer distances. This is why it is almost exclusively used for board-local purposes.
- **1-Wire:** One Dallas 1-Wire bus connector. 1-Wire consists of only 1 wire plus a ground wire, hence the name. Devices consumes data and power from the same wire. They power themselves using small capacitors when no data is being sent. The bus is an addressable micro local area network (LAN) with unique 64-bit addresses. It is similar to I_2C, but slower. The protocol supports communication over longer distances. This makes it useful for external sensors not local to the board.

The GPIO, UART, and SPI ports follows the Peripheral Module (PMOD) form factor. You can connect plug-and-play devices without soldering. Many vendors supply PMOD devices implementing many different hardware functions. Examples include network interfaces, analog-digital converters, displays or servor motor connectors. Others provide sensors such as accelerometers, gyroscopes, and global positioning system (GPS) receivers. Users can also develop your own PMOD by following the specification.

Network and Connectivity Besides providing power the USB supplies serial and debugging connections. It has an integrated serial interface and a Joint Test Action Group (JTAG) interface. The serial interface accesses applications running on the board. Developers can use the JTAG interface to connect a debugger. The debugger will have full external access to the CPU. This allows for stepping through CPU instructions on an hardware level. There is also an external 2x10 pin JTAG connector on the board itself. Here you can attach special JTAG hardware debuggers.

The board has built-in Wi-Fi. The chip provides standard 802.11n wireless access for the 2.4 Ghz band with speeds of up to 150 Mbps. With its power saving mechanisms it is useful for temporary ad-hoc mesh networking.

Users and developers can also attach external hardware providing more connectivity options. Examples include ethernet, bluetooth, or custom radio PMODs.

(b) Software Stack

We have selected Erlang as the main development language and runtime environment. The Erlang virtual machine (VM) comes with a standard library and a set of design principles, OTP. It provides a distributed, fault-tolerant soft real-time environment.

Erlang itself is a high-level functional programming language. It has immutable data and pattern matching suited to advanced application development. Erlang has many useful primitives for implementing networking protocols. You can also interface with C making it a good fit for embedded development.

Our goal is to run Erlang on real bare metal hardware. That means without a separate operating system in-between the hardware and the application. We have opted to use The Real-Time Executive for Multiprocessor Systems (RTEMS). It is a Real Time Operating System (RTOS) as a library. RTEMS implements common operating system application programming interfaces (API). One such API is Portable Operating System Interface (POSIX) which Erlang uses. When compiling Erlang together with RTEMS, you get one executable. The boot loader then executes that executable as the OS itself. It has everything needed to interface with the hardware built-in. For application developers it looks like a normal Erlang VM.

One definition of GRiSP is thus the combination of Erlang together with RTEMS. This makes it possible to run the VM on various hardware boards. The GRiSP base is one such board.

(c) Dissemination Activities

Other partners in the Lightkone project will use the platform as a base for prototyping.

We have made several talks at various conferences presenting the GRiSP project. We are showing why it is beneficial for IoT applications. We have also been giving hands-on tutorials on using the GRiSP board.

The project itself is open source. It is also non-profit in the sense that Peer Stritzinger GMBH develops, manufactures and sells the hardware at production cost. The motivation for this is in expanding the community taking advantage of GRiSP for prototyping projects. Both by the Lightkone partners and by hobbyists. This will increase the uptake of the platform. In turn it increases knowledge of advances in edge computing and the Lightkone project itself.

The Lightkone consortium also believes that GRiSP can become a useful product in the context of teaching activities. In particular for advanced courses related with embedded systems and, potentially in combination with the Yggdrasil framework discussed earlier in this document, in advanced networking and distributed systems courses.

(d) State of the Art

There are many hardware projects with accompanying development boards in the market . The most popular, with a very well-established community, is the Arduino project [1] that focuses on resource constrained prototyping boards. Other alternatives include the BeagleBoard, Raspberry Pi or Raspberry Pi Zero [7], among others. Commonly, resource

contrained devices, that are usually programmed using C-like languages, lack the capacities to implement a full network stack for interactions with Internet services. On the other hand, the boards that are able to implementent network interfaces, usually run an instance of a general-purpose Operating System and are not energy-efficient.

Nerves [2] provides an Erlang VM based environment suited to embedded development. It is targeting larger devices that run Linux as the base OS. Supported platforms include BeagleBone and Raspberry Pi. GRiSP sets itself in a point of the design space in between the choices made by these platforms.

(e) Discussion

GRiSP provides a trade-off between simpler smaller devices and larger more powerful devices. It is powerful enough to run advanced applications implemented in Erlang. It is also small enough to serve as a compact base for embedded IoT devices.

The possibility to use a high-level language and plug-and-play hardware makes it attractive. Applications can be rapidly prototyped in Erlang, combined with off-the-shelf components. They come in the form of PMODs and other peripheral hardware.

Due to the pluggable nature of the GRiSP board, it is not always suited as a target for final embedded devices. It is always cheaper to manufacture larger amounts of devices with custom hardware. Once you settle on the requirements and design many optimizations are possible. Manufacturing custom hardware makes it possible reduce the size of the device. You can also make it more efficient and lower power consumption with custom hardware (and by removing any hardware that is not required for a concrete use case).

In the future we want to continue the development the software stack. To get new features, we want to keep up with the latest versions of Erlang. We also want to add support for the crypto libraries that come with Erlang. We can then build secure IoT applications and other edge-enabled applications, and support protocols with built-in security.

Because other Lightkone partners use GRiSP, we expect to continue to make changes. Either new Erlang features or developing low-level libraries to access certain hardware. We also aim to target different hardware platforms other that GRiSP base. This will need more support from our build tools. Another interesting venue for future development would be to integrate Yggdrasil into the GRiSP ecosystem. This can be achieved by either extending the Yggdrasil low level library for it to become compatible with the interface for radio devices exposed by RTEMS. Another approach, would be by wrapping the current prototype of Yggdrasil with an Erlang API, enabling edge applications written in Erlang to take advantage of the mechanisms exposed by Yggdrasil, namely some of the implementations of distributed membership, dissemination, and aggregation protocols currently featured in the framework.

4 Secondary Deliverable Contributions

4.1 Non Uniform CRDTs

A key aspect in building the necessary tools and support for enabling a new generation of edge-enabled applications is to allow the inter-operation of different system components running in highly distinct edge environments. As discussed in deliverable D3.1,

the Lightkone consortium envisions two key aspects for achieving this purpose. The first is to define adequate data models that can ease the transference of application state and computations among different components of an edge-enabled distributed application; the second is related to the definition of clear and standard APIs and simple support services that can ease the integration and communication between the different components of the application.

CRDTs leveraging Non-Uniform Replication schemes is a first and decisive step in devising an adequate data model for edge-enabled large-scale distributed applications, as these data types allow for relevant application state to be encapsulated within a well defined data type, with clear semantics. This allows to perform efficient data replication and data transference, by avoiding to transfer large quantities of data. In particular, only data that is relevant for performing a given step of a large distributed computation or task. Furthermore, in the particular case of replication, a key technique in the design of efficient and reliable distributed systems, is to allow replicas to synchronize under the eventual consistency model, hence avoiding costly (both in terms of latency and processing power) synchronization steps in the critical path of user operations.

While this contribution is fully detailed in D3.1, in this document, for completeness, we provide a brief overview of this contribution, and discuss how these novel data types can be used for instance, to improve the performance of data aggregation algorithms across multiple edge scenarios. This section is structured as follows. § 4.1 (a) presents the system model assumed in this contribution, introducing the relevant concepts of Non-Uniform Replication and Non-Uniform Eventual Consistency. § 4.1 (b) presents a brief overview on the design of CRDTs leveraging Non-Uniform Replication. Finally, § 4.1 (c) briefly overview relevant related work and § 4.1 (d) discusses future directions for the adoption of Non-Uniform Replication as a model for supporting large-scale edge-enabled applications.

This work was originally published in [25]. For completeness and convenience, the full content of the original paper is provided in this document in Appendix *B*.

(a) System Model & Relevant Concepts

There are two key (novel) concepts for understanding the design and implementation of Non-Uniform operation-based CRDTs: *i*) Non-Uniform Replication and *ii*) Non-Uniform Eventual Consistency. Here we provide a condensed overview of these concepts as introduced in [25]. The interested reader is referred to the full publications (see Appendix *B*) for details and proofs. This work is also further detailed in Deliverable D3.1.

Non-Uniform Replication We consider an asynchronous distributed system composed by n nodes. Without loss of generality, we assume that the system replicates a single object. The object has an interface composed by a set of read-only operations, $\mathscr{Q}$, and a set of update operations, $\mathscr{U}$. Let $\mathscr{S}$ be the set of all possible object states, the state that results from executing operation o in state $s \in \mathscr{S}$ is denoted as $s \bullet o$. For a read-only operation, $q \in \mathscr{Q}$, $s \bullet q = s$. The result of operation $o \in \mathscr{Q} \cup \mathscr{U}$ in state $s \in \mathscr{S}$ is denoted as $o(s)$ (we assume that an update operation, besides modifying the state, can also return some result).

We denote the state of the replicated system as a tuple $(s_1, s_2, \ldots, s_n)$, with s_i the state

of the replica *i*. The state of the replicas is synchronized by a replication protocol that exchanges messages among the nodes of the system and updates the state of the replicas.

A system is in a quiescent state for a given set of executed operations if the replication protocol has propagated all messages necessary to synchronize all replicas.

Definition 1 (Equivalent state) *Two states, s_i and s_j, are* equivalent*, $s_i \equiv s_j$, iff the results of the execution of any sequence of operations in both states are equal, i.e., $\forall o_1, \ldots, o_n \in \mathcal{Q} \cup \mathcal{U}, o_n(s_i \bullet o_1 \bullet \ldots \bullet o_{n-1}) = o_n(s_j \bullet o_1 \bullet \ldots \bullet o_{n-1})$.*

Note that this property does not require the internal state of the replicas to be the same, but only that the replicas always return the same results for any executed sequence of operations.

We propose to relax this property by requiring only that the execution of read-only operations return the same value. This property is named *observable equivalence* and is defined as:

Definition 2 (Observable equivalent state) *Two 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 in both states is equal, i.e., $\forall o \in \mathcal{Q}, o(s_i) = o(s_j)$.*

As read-only operations do not affect the state of a replica, the results of the execution of any sequence of read-only operations in two observable equivalent states will also be the same. We now define a non-uniform replication system as one that guarantees only that replicas converge to an observable equivalent state.

Definition 3 (Non-uniform replicated system) *We say that a replicated system is non-uniform if the replication protocol guarantees that in a quiescent state, the state of any two replicas is observable equivalent, i.e., in the quiescent state $(s_1, \ldots, s_n)$, we have $s_i \overset{o}{\equiv} s_j, \forall s_i, s_j \in \{s_1, \ldots, s_n\}$.*

Non-Uniform Eventual Consistency For any given execution, with O the operations of the execution, we say a replicated system provides *eventual consistency* iff in a quiescent state: (i) every replica executed all operations of O; and (ii) the state of any pair of replicas is equivalent.

For any given execution, with O the operations of the execution, we say 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. 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 $s^0 \bullet O \overset{o}{\equiv} s^0 \bullet O_{core}$. We define the set of operations that are irrelevant to the final state of the replicas as follows: $O_{masked} \subseteq O$ is a set of masked operations of O iff $s^0 \bullet O \overset{o}{\equiv} s^0 \bullet (O \setminus O_{masked})$.

(b) Non-Uniform Operation-based CRDTs

CRDTs [78] are data-types that can be replicated, modified concurrently without coordination, and guarantee the eventual consistency of replicas given that all updates propagate to all replicas. We now present the design of two useful operation-based CRDTs [78] that

Algorithm 1 Replication algorithm for non-uniform eventual consistency

```
1:  S : state: initial s^0                                              ▷ Object state
2:  log_recv : set of operations: initial {}
3:  log_local : set of operations: initial {}                           ▷ Local operations not propagated
4:
5:  EXECOP(op): void                                                    ▷ New operation generated locally
6:    log_local = log_local ∪ {op}
7:    S = S • op
8:
9:  OPSTOPROPAGATE(): set of operations   ▷ Computes the local operations that need to be propagated
10:   ops = maskedForever(log_local, S, log_recv)
11:   log_local = log_local \ ops
12:   opsImpact = hasObservableImpact(log_local, S, log_recv)
13:   opsPotImpact = mayHaveObservableImpact(log_local, S, log_recv)
14:   return opsImpact ∪ opsPotImpact
15:
16: SYNC(): void                                  ▷ Propagates local operations to remote replicas
17:   ops = opsToPropagate()
18:   compactedOps = compact(ops)                                       ▷ Compacts the set of operations
19:   mcast(compactedOps)
20:   log_coreLocal = {}
21:   log_local = log_local \ ops
22:   log_recv = log_recv ∪ ops
23:
24: ON RECEIVE(ops): void                                               ▷ Process remote operations
25:   log_recv = log_recv ∪ ops
26:   S = S • ops
```

adopt the non-uniform replication model. Unlike most operation-based CRDT designs, we do not assume that the system propagates operations in a causal order. These designs were inspired by the state-based computational CRDTs proposed by Navalho *et al.* [68], which also allow replicas to diverge in their quiescent state. These CRDTs can be leveraged to support specific data aggregation operators in an distributed and efficient way in the edge.

Our design follows a general replication strategy that strives to minimize the amount of operations that have to be replicated. The algorithm is depicted in Algorithm 1. The key idea is to avoid propagating operations that are part of a masked set. The challenge is to achieve this by using only local information, which includes only a subset of the executed operations.

This is fundamentally captured by the function *sync*, which is called to propagate local operations to remote replicas. It uses the function *opsToPropagate*, that addresses the key challenge of deciding which operations need to be propagated to other replicas. To this end, we divide the operations in four groups.

First, the *forever masked* operations, which are operations that will remain in the set of masked operations independently of the operations that might be executed in the future.

Second, the *core* operations (*opsImpact*, line 12), as computed locally. These operations need to be propagated, as they will (typically) impact the observable state at every replica.

Third, the operations that might impact the observable state when considered in com-

bination with other non-core operations that might have been executed in other replicas.

Fourth, the remaining operations that might impact the observable state in the future, depending on the evolution of the observable state.

A detailed explanation of this algorithm can be found in [25]. In the following we detail the design of two Non-Uniform CRDTs. The design of these new data types is presented by materializing the generic protocol discussed here.

Top-K with removals NuCRDT In this section we present the design of a non-uniform top-K CRDT. The data type allows access to the top-K elements added and can be used, for example, for maintaining the leaderboard in online games. The proposed design could be adapted to define any CRDT that filters elements based on a deterministic function by replacing the *topK* function by another filter function.

The semantics of the operations defined in the top-K CRDT is the following. The *add(el,val)* operation adds a new pair to the object. The *rmv(el)* operation removes any pair of *el* that was added by an operation that happened-before the *rmv* (note that this includes non-core add operations that have not been propagated to the source replica of the remove). This leads to an *add-wins* policy [78], where a remove has no impact on concurrent adds. The *get()* operation returns the top-K pairs in the object, as defined by the function *topK* used in the algorithm.

Algorithm 2 presents a design that implements this semantics. The prepare-update *add* operation generates an effect-update *add* that has an additional parameter consisting in a timestamp $(replicaid, val)$, with *val* a monotonically increasing integer. The prepare-update *rmv* operation generates an effect-update *rmv* that includes an additional parameter consisting in a vector clock that summarizes add operations that happened before the remove operation. To this end, the object maintains a vector clock that is updated when a new add is generated or executed locally. Additionally, this vector clock should be updated whenever a replica receives a message from a remote replica (to summarize also the adds known in the sender that have not been propagated to this replica).

Besides this vector clock, *vc*, each object replica maintains: (i) a set, *elems*, with the elements added by all *add* operations known locally (and that have not been removed yet); and (ii) a map, *removes*, that maps each element *id* to a vector clock with a summary of the add operations that happened before all removes of *id* (for simplifying the presentation of the algorithm, we assume that a key absent from the map has associated a default vector clock consisting of zeros for every replica).

The execution of an *add* consists in adding the element to the set of *elems* if the add has not happened before a previously received remove for the same element – this can happen as operations are not necessarily propagated in causal order. The execution of a *rmv* consists in updating *removes* and deleting from *elems* the information for adds of the element that happened before the remove.

We now analyze the code of these functions.

Function MASKEDFOREVER computes: the local adds that become masked by other local adds (those for the same element with a lower value) and removes (those for the same element that happened before the remove); the local removes that become masked by other removes (those for the same element that have a smaller vector clock). In the latter case, it is immediate that a remove with a smaller vector clock becomes irrelevant after executing the one with a larger vector clock. In the former case, a local add for an element is masked by a more recent local add for the same element but with a larger

Algorithm 2 Top-K NuCRDT with removals

1: $elems$: set of $\langle id, score, ts\rangle$: initial $\{\}$
2: $removes$: map $id \mapsto vectorClock$: initial []
3: vc : $vectorClock$: initial []
4:
5: GET() : set
6: **return** $\{\langle id, score\rangle : \langle id, score, ts\rangle \in topK(elems)\}$
7:
8: **prepare** ADD$(id, score)$
9: **generate** $add(id, score, \langle getReplicaId(), ++vc[getReplicaId()]\rangle)$
10:
11: **effect** ADD$(id, score, ts)$
12: **if** $removes[id][ts.siteId] < ts.val$ **then**
13: $elems = elems \cup \{\langle id, score, ts\rangle\}$
14: $vc[ts.siteId] = max(vc[ts.siteId], ts.val)$
15:
16: **prepare** RMV(id)
17: **generate** $rmv(id,vc)$
18:
19: **effect** RMV(id, vc_{rmv})
20: $removes[id] = pointwiseMax(removes[id], vc_{rmv})$
21: $elems = elems \setminus \{\langle id_0, score, ts\rangle \in elem : id = id_0 \wedge ts.val \leq vc_{rmv}[ts.siteId]\}$
22:
23: MASKEDFOREVER$(log_{local}, S, log_{recv})$: set of operations
24: $adds = \{add(id_1, score_1, ts_1) \in log_{local} :$
25: $(\exists add(id_2, score_2, ts_2) \in log_{local} : id_1 = id_2 \wedge score_1 < score_2 \wedge ts_1.val < ts_2.val) \vee$
26: $(\exists rmv(id_3, vc_{rmv}) \in (log_{recv} \cup log_{local}) : id_1 = id_3 \wedge ts_1.val \leq vc_{rmv}[ts_1.siteId]\}$
27: $rmvs = \{rmv(id_1, vc_1) \in log_{local} : \exists rmv(id_2, vc_2) \in (log_{local} \cup log_{recv}) : id_1 = id_2 \wedge vc_1 < vc_2\}$
28: **return** $adds \cup rmvs$
29:
30: MAYHAVEOBSERVABLEIMPACT$(log_{local}, S, log_{recv})$: set of operations
31: **return** $\{\}$ ▷ This case never happens for this data type
32:
33: HASOBSERVABLEIMPACT$(log_{local}, S, log_{recv})$: set of operations
34: $adds = \{add(id_1, score_1, ts_1) \in log_{local} : \langle id_1, score_1, ts_1\rangle \in topK(S.elems)\}$
35: $rmvs = \{rmv(id_1, vc_1) \in log_{local} : (\exists add(id_2, score_2, ts_2) \in (log_{local} \cup log_{recv}) :$
36: $\langle id_2, score_2, ts_2\rangle \in topK(S.elems \cup \{\langle id_2, score_2, ts_2\rangle\}) \wedge id_1 = id_2 \wedge ts_2.val \leq vc_1[ts_2.siteId])\}$
37: **return** $adds \cup rmvs$
38:
39: COMPACT(ops): set of operations
40: **return** ops ▷ This data type does not require compaction

value as it is not possible to remove only the effects of the later add without removing the effect of the older one. A local add also becomes permanently masked by a local or remote remove that happened after the add.

Function MAYHAVEOBSERVABLEIMPACT returns the empty set, as for having impact on any observable state, an operation also has to have impact on the local observable state by itself.

Function HASOBSERVABLEIMPACT computes the local operations that are relevant for computing the top-K. An add is relevant if the added value is in the top; a remove is relevant if it removes an add that would be otherwise in the top.

Top Sum NuCRDT We now present the design of a non-uniform CRDT, Top Sum, 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).

The semantics of the operations defined in the Top Sum object is the following. The *add(id, n)* update operation increments the value associated with *id* by *n*. The *get()* read-only operation returns the top-K mappings, $id \rightarrow value$, as defined by the *topK* function (similar to the Top-K NuCRDT).

This design is challenging, as it is hard to know which operations may have impact in the observable state. For example, consider a scenario with two replicas, where the value of the last element in the top is 100. If the known score of an element is 90, an add of 5 received in one replica may have impact in the observable state if the other replica has also received an add of 5 or more. One approach would be to propagate these operations, but this would lead to propagating all operations.

To try to minimize the number of operations propagated we use the following heuristic inspired by the demarcation protocol and escrow transactions [16, 71]. For each id that does not belong to the top, we compute the difference between the smallest value in the top and the value of the id computed by operations known in every replica – this is how much must be added to the id to make it to the top: let d be this value. If the sum of local adds for the id does not exceed $\frac{d}{num.replicas}$ in any replica, the value of id when considering adds executed in all replicas is smaller than the smallest element in the top. Thus, it is not necessary to propagate add operations in this case, as they will not affect the top.

Algorithm 3 presents a design that implements this approach. The state of the object is a single variable, *state*, that maps identifiers to their current values. The only prepare-update operation, *add*, generates an effect-update *add* with the same parameters. The execution of an effect-update *add(id, n)* simply increments the value of *id* by *n*.

Function MASKEDFOREVER returns the empty set, as operations in this design can never be forever masked.

Function MAYHAVEOBSERVABLEIMPACT computes the set of *add* operations that can potentially have an impact on the observable state, using the approach previously explained.

Function HASOBSERVABLEIMPACT computes the set of *add* operations that have their corresponding *id* present in the top-K. This guarantees that the values of the elements in the top are kept up-to-date, reflecting all executed operations.

Function COMPACT takes a set of *add* operations and compacts the *add* operations that affect the same identifier into a single operation. This reduces the size of the messages sent through the network and is similar to the optimization obtained in delta-based CRDTs [10].

(c) State of the Art

Replication: A large number of replication protocols have been proposed in the last decades [11, 35, 59–61, 76, 92]. Regarding the contents of the replicas, these protocols

Algorithm 3 Top Sum NuCRDT

```
1:  state : map id ↦ sum: initial []
2:
3:  GET() : map
4:    return topK(state)
5:
6:  prepare ADD(id, n)
7:    generate add(id, n)
8:
9:  effect ADD(id, n)
10:   state[id] = state[id] + n
11:
12: MASKEDFOREVER(log_local, S, log_recv): set of operations
13:   return {}                          ▷ This case never happens for this data type
14:
15: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
16:   top = topK(S.state)
17:   adds = {add(id, _) ∈ log_local : s = sum_val({add(i, n) ∈ log_local : i = id})
18:     ∧ s ≥ ((min(sum(top)) − (S.state[id] − s)) / getNumReplicas())}
19:   return adds
20:
21: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
22:   top = topK(S.state)
23:   adds = {add(id, _) ∈ log_local : id ∈ ids(top)}
24:   return adds
25:
26: COMPACT(ops): set of operations
27:   adds = {add(id, n) : id ∈ {i : add(i, _) ∈ ops} ∧ n = sum({k : add(id_1, k) ∈ ops : id_1 = id})}
28:   return adds
```

can be divided in those providing full replication, where each replica maintains the full database state, and partial replication, where each replica maintains only a subset of the database state.

Full replication strategies allow operations to concurrently modify all replicas of a system and, assuming that replicas are mutually consistent, improves availability since clients may query any replica in the system and obtain an immediate response. While this improves the performance of read operations, update operations now negatively affect the performance of the system since they must modify every replica which severely affects middle-scale to large-scale systems in geo-distributed settings. This model also has the disadvantage of limiting the system's total capacity to the capacity of the node with fewest resources.

Partial replication [12, 32, 77, 86] addresses the shortcomings of full replication by having each replica store only part of the data (which continues being replicated in more than one node). This improves the scalability of the system but since each replica maintains only a part of the data, it can only locally process a subset of queries. This adds complexity to the query processing, with some queries requiring contacting multiple replicas to compute their result. In our work we address these limitations by proposing a model where each replica maintains only part of the data but can reply to any query.

Despite of adopting full or partial replication, replication protocols enforce strong consistency [30, 61, 65], weak consistency [11, 34, 59, 60, 92] or a mix of these consistency models [58, 80]. In this paper we show how to combine non-uniform replication

with eventual consistency. An important aspect in systems that adopt eventual consistency is how the system handles concurrent operations. CRDTs have been proposed as a technique for addressing such challenge.

CRDTs: Conflict-free Replicated Data Types [78] are data types designed to be replicated at multiple replicas without requiring coordination for executing operations. CRDTs encode merge policies used to guarantee that all replicas converge to the same value after all updates are propagated to every replica. This allows an operation to execute immediately on any replica, with replicas synchronizing asynchronously. Thus, a system that uses CRDTs can provide low latency and high availability, despite faults and network latency. With these guarantees, CRDTs are a key building block for providing eventual consistency with well defined semantics, making it easier for programmers to reason about the system evolution.

When considering the synchronization process, two main types of CRDTs have been proposed: state-based CRDT, where replicas synchronize pairwise, by periodically exchanging the state of the replicas; and operation-based CRDTs, where all operations need to be propagated to all replicas.

Delta-based CRDTs [9] improve upon state-based CRDTs by reducing the dissemination cost of updates, sending only a delta of the modified state. This is achieved by using *delta-mutators*, which are functions that encode a delta of the state. Linde et. al [90] propose an improvement to delta-based CRDTs that further reduce the data that need to be propagated when a replica first synchronizes with some other replica. This is particularly interesting in peer-to-peer settings, where the synchronization partners of each replica change frequently. Although delta-based CRDTs reduce the network bandwidth used for synchronization, they continue to maintain a full replication strategy where the state of quiescent replicas is equivalent.

Computational CRDTs [68] are an extension of state-based CRDTs where the state of the object is the result of a computation (e.g. the average, the top-K elements) over the executed updates. As with the model we propose in this paper, replicas do not need to have equivalent states. This work extends the initial ideas proposed in computational CRDTs in several aspects, including the definition of the non-uniform replication model, its application to operation-based eventual consistency and the new data type designs.

(d) Discussion

The current design of Non-Uniform CRDTs paves the way towards an adequate data representation model for supporting the operation of complex edge-enabled applications. This is achieved by minimizing data transference and hence synchronization costs among replicas, while providing support to perform efficient operations over potentially large collections of data.

The current prototype implementation of Non-Uniform CRDTs only addresses very specific aggregation functions, however this demonstrates the feasibility of building specific Non-Uniform CRDTs to support other aggregation operators. Another relevant aspect that we plan to address in the context of Non-Uniform CRDTs is related with the support of multiple operations (i.e, queries) over the same data collection. The current prototype design strives to find the minimal data that has to be synchronized among replicas in order to ensure that a particular query achieves non-uniform eventual consis-

tency. Generalizing this design will allow to move towards the support of general purpose computations with low synchronization and data transference overhead. Aggregation operators are an essential primitive for achieving general purpose computations. However, to achieve this, there must be ways to compose these computations into more complex ones. The current design of Non-Uniform CRDTs do not offer a simple API to compose multiple computations over multiple CRDTs. Additional effort has to be conducted to find the most adequate way to offer such support.

We plan to address these research and development questions in future work, as well as to integrate these new types of CRDTs into some of the tools described in this report (Yggdrasil, Legion, and LASP).

4.2 Enriching Consistency at the Edge

Services running on the core of systems, such as distributed and geo-replicated storage systems [11, 52] and online services such as Facebook or Twitter, have public APIs to enable an easy integration with applications. These applications usually run on clients and can be part of sophisticated edge architectures, with components scattered throughout many different environments. However, the developers who design these applications may be required to handle specific anomalies allowed by the consistency models exposed by these services running in the core of the network. This can be a complex task, particularly in the case of online services that provide little or no information about the consistency provided by these services. This forces application developers to reason on how to enforce the semantics of their applications when the core of the system offers little consistency guarantees.

To overcome this challenge, and to pave the way for easier integration of different components in edge-enabled applications, we have developed a transparent middleware that can operate between the edge component of an application and the service that executes at the core of the system (or even at another layer of the edge closer to the center of the network). Our solution enables a fine-grained control over the session guarantees that comprise the consistency semantics provided by such systems, without having to gain access or modify the implementation of those services. To demonstrate the feasibility of our approach we have applied it for the Facebook public API and the Redis datastore, allowing to have fine-grained control over the consistency semantics observed by clients with a small local storage and modest latency overhead.

While this contribution focus on enriching the session guarantees from the standpoint of a single client session, this technique is particularly useful to allow the integration of heavy edge storage system and third party services into edge-enabled applications. Due to this, additional details on this contribution can be found in the deliverable D6.1 produced by Work Package 6 of the Lightkone consortium. This work was original published in [41]. For the readers convenience the complete text of the original publication is reproduced in Appendix.

(a) System Model

Our goal is to enrich consistency guarantees provided by third-party applications, allowing easier integration in more complex (edge-enabled) applications. In particular, the application developer may choose to have individual session guarantees (read your write,

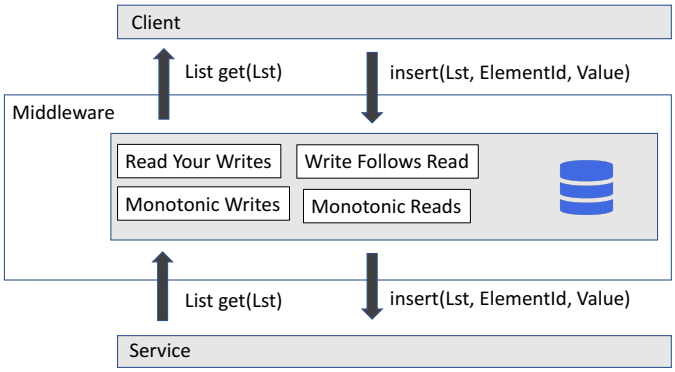


Figure 4.1: Middleware Architecture

monotonic reads, monotonic writes, and writes follows reads) as well as combinations of these properties (in particular, all four session guarantees corresponds to causality [24]). To achieve this, we provide a library that can be easily attached to a client application (operating as a middleware layer), allowing to enrich the semantics exposed through the system public API. We focus on services that expose a data model based on key-value stores, where data objects can be accessed through a key, and that associate a list of objects to each key. We observe that this data model is prevalent in online social network services, particularly since they share concepts, such as user feeds and comment lists. In particular, we target services where the API provides two fundamental operations to manipulate the list of objects associated with a given key: an insert operation to append a new object to the first position of the list, and a get operation that exposes the first N elements of the list (i.e., the most recent N elements).

Since we access these services through their public APIs, we need to view the service implementation as a black box, meaning that no assumptions are made regarding their internal operation. Furthermore, we design our protocols without making any assumption regarding the consistency guarantees provided through the public service API. Previous work has shown a high prevalence of session guarantees violations in services exposing public APIs [41].

Our middleware solution leverages on a set of algorithms that require storing meta-data alongside the data, which can be difficult to do when accessing services as black boxes, namely when the service has no support for including user managed metadata. In this case, we need to encode this meta-data as part of the data itself.

In order to arbitrate an ordering among operations issued by the local client and other remote clients, our Middleware has the need to have an approximate estimate of the current time. To achieve this, two options are available. If the service has a specific call in its public API that exposes the time in the server, such call can directly be used by our system. Otherwise, if the service exposes a REST API (which is typical in many services) a simple REST call can be performed to the service, and the server time can be extracted from a standard HTTP response header (called *Date*). Note that, even though it is desirable that this estimate is synchronized across clients, we do not require either clock or clock rate synchronization for correctness.

(b) Architecture

Our solution is materialized in a library implementing a middleware layer. The architecture of this middleware solution is depicted in Figure 4.1. Our system consists of a thin layer that runs on the client side and intercepts every call made by the client application over the service, mediating access to the service. In particular, our layer is responsible for contacting the service on behalf of the client application, process the responses returned by the service and generate responses to the client applications according with the session guarantees being enforced.

Our system can be configured by the client application developer to enforce any combination of the individual session guarantees (as defined by Terry et. al [85]), namely: *i*) read your writes, *ii*) monotonic reads, *iii*) monotonic writes, and *iv*) writes follows reads. In order to enforce these guarantees, our system is required to maintain information regarding previous operations executed by the client application, namely previous writes that were issued or previous values that were observed by the client.

(c) Enriching Consistency

As previously mentioned, our system intercepts each request performed by the client application, executes the request in the service, and then processes the answer generated by the service to provide a (potentially different) answer to the client application. This answer is computed based on a combination of the internal state that records the previous operations that were run by that particular client, and the actual response that was returned by the service.

Tracking application activity. In order to keep track of user activity, our system maintains in memory a set of data structures for each part of the service state that is accessed by the application. These data structures are updated according to the activity of the applications (i.e., the operations that were invoked) and the state that is returned by the service. These data structures are: *i*) the *insertSet*, which stores the elements inserted by the client and *ii*) the *localView*, which stores the elements returned to the client.

Enforcing session guarantees. Enforcing session guarantees entails achieving two complementary aspects. First, and depending on the session guarantees being enforced, some additional metadata must be added when inserting operations. As mentioned, this metadata can be either added to a specialized metadata field (if the API exposed by the service allows this) or directly encoded within the body of the element being added to the list. Such metadata has to be extracted by our library when retrieving the elements of a list, thus ensuring transparency towards client applications. Second, our system might be required to either remove or add elements to the list that is returned by the service when the application issues an operation to obtain the current service state, in order to ensure that the intended session guarantees are not violated.

We now discuss the concrete guarantees that can be enforced by our solution. The interested reader is referred to [41] or to the deliverable D6.1 to obtain the full details regarding the algorithms employed to achieve this. We also note, that these individual guarantees can be combined (i.e, the middleware can provide more than one of these guarantees simultaneously).

Read Your Writes The Read Your Writes (RYW) session guarantee requires that, in a session, any read observes all writes previously executed by the same client. More

precisely, for every set of insert operations W made by a client c over a list L in a given session, and set S of elements from list L returned by a subsequent get operation of c over L, we say that RYW is violated if and only if $\exists x \in W : x \notin S$.

This definition, however, does not consider the case where only the N most recent elements of a list are returned by a get operation. In this case, some writes of a given client may not be present in the result if more than N other insert operations have been performed (by client c or any other client). Considering that the list must hold the most recent writes, a RYW anomaly happens when a get operation returns an older write performed by the client but misses a more recent one. More formally, given two writes x, y over list L executed in the same client session, where x was executed before y, an anomaly of RYW happens in a get that returns S when $\exists x, y \in W : x \prec y \wedge y \notin S \wedge x \in S$.

Monotonic Reads This session guarantee requires that all writes reflected in a read are also reflected in all subsequent reads performed by the same client. To define this in our scenario where a truncated list of N recent elements is returned, we say that Monotonic Reads (MR) is violated when a client c issues two read operations that return sequences S_1 and S_2 (in that order) and the following property holds: $\exists x, y \in S_1 : x \prec y \; in \; S_1 \wedge y \notin S_2 \wedge x \in S_2$, where $x \prec y$ means that element x appears in S_1 before y.

Monotonic Writes This session guarantee requires that writes issued by a given client are observed in the order in which they were issued by clients. More precisely, if W is a sequence of write operations issued by client c up to a given instant, and S is a sequence of write operations returned in a read operation by any client, a Monotonic Writes (MW) anomaly happens when the following property holds, where $W(x) \prec W(y)$ denotes x precedes y in sequence W: $\exists x, y \in W : W(x) \prec W(y) \wedge y \in S \wedge (x \notin S \vee S(y) \prec S(x))$.

However, this definition needs to be adapted for the case where only N elements of a list are returned by a get operation. In this case, some session sequences may be incomplete, because older elements of the sequence may be left out of the truncated list of N returned elements. Thus, we consider that older elements are eligible to be dropped from the output, provided that we ensure that there are no gaps in the session subsequences and that the write order is respected, before returning to the client. Formally, we can redefine MW anomalies as follows, given a sequence of writes W in the same session, and a sequence S returned by a read: $(\exists x, y, z \in W : W(x) \prec W(y) \prec W(z) \wedge x \in S \wedge y \notin S \wedge z \in S) \vee (\exists x, y \in W : W(x) \prec W(y) \wedge S(y) \prec S(x))$.

Write Follows Read This session guarantee requires that the effects of a write observed in a read by a given client always precede the writes that the same client subsequently performs (Note that although this anomaly has been used to exemplify causality violations [11, 59], any of the previous anomalies represent a different form of a causality violation [85]). To formalize this definition, and considering that the service only returns at most N elements in a list, if S_1 is a sequence returned by a read invoked by client c, w a write performed by c after observing S_1, and S_2 is a sequence returned by a read issued by any client in the system; a violation of the Write Follows Read (WFR) anomaly happens when: $w \in S_2 \wedge \exists x, y \in S_1 : x \prec y \; in \; S_1 \wedge y \notin S_2 \wedge x \in S_2$.

(d) State of the Art

The closest related work can be found in recent proposals that also leverage a middleware layer that can mediate access to a storage system in order to upgrade their respective consistency guarantees [15, 18].

In particular, Bailis et al. [15] proposed a system called "bolt-on causal consistency" to offer causal consistency on top of eventually consistent data stores. There are two main distinctions between bolt-on causal consistency and our proposal: first, we provide a fine-grained choice of which session guarantees the programmer intends the system to provide, and only pay a performance penalty that is associated with enforcing those guarantees. Second, they assume the underlying system offers a general read/write storage interface, which gives significant more flexibility in terms of the system design than in our proposal.

The other closely related system is the one proposed by Bermbach et al. [18], which is also based on a generic storage interface, such as the ones exported by S3, DynamoDB, or SimpleDB, in contrast to our focus on high level service APIs. While they also provide fine-grained session guarantees chosen by the programmer, they limit these to Monotonic Reads and Read Your Writes.

(e) Discussion

This work enables the client of an external service (either a distributed data storage system or some service accessed through a public API) to observe states of the application that respect some or a subset (eventually all) of the session guarantees. This is achieved through the use of a middleware layer that intercepts the calls performed by the client to the service and interacts with the service by the client. This middleware is then reponsible for transparently transforming the reply of the service that is returned to the client to enforce the required session guarantees.

This is a powerful abstraction that can simplify the integration of independent services within a complex application, namely an edge application, enabling developers to design their applications without being concerned with potential session guarantees that are not enforced by that system consistency model.

We note however that currently our approach is focused on the interfaces that are commonly exposed by social network applications, namely that of a list where different users can add (i.e, append) elements to the list and read the N most recent entries in the list. Expanding this approach to deal with more general data models is an open challenge. We also note that this approach is fundamentally useful for assisting in the integration of existing services to edge-enabled applications. Services that already offer causal consistency guarantees (such as AntidoteDB described in D6.1) or that expose data models based on CRDTs (see D3.1) will not require the use of this approach, and hence, avoid the storage and latency overheads introduced by it.

5 Relationship of Results with Industrial Use Cases

We now discuss the relationship with the Industrial Use Cases previously reported in D2.1. We restrict our discussion to the use cases that can at least partially be tackled by

the contributions reported in this document. We refer the interested reader to the D2.1 report for obtaining full details on these use cases.

The main contributions reported in this deliverable that can be explored by the Lightkone consortium are the GRiSP platform and the Yggdrasil framework. These two contributions both pave the way for a new generation of edge-enabled applications. This is achieved by enabling the development of novel hardware prototypes and experimentation in an expedite way and also, by offering different programming abstractions to develop distributed protocols and applications leveraging on large numbers of small devices.

The combination of these two contributions will allow us to tackle some of the challenges that arise in use cases such as the *smart metering gateway* (provided by Peer Stritzinter GMBH) and the *agriculture sensing analytics* (provided by Gluk). Both of these use cases will benefit from having small devices with wireless antennas, that coordinate the monitoring and processing of data directly on the edge. The GRiSP platform will enable to prototype new devices for this end, while the programming environment offered by the GRiSP software stack will allow to develop applications in these contexts. In fact, Peer Strinzinter GMBH offers GRiSP for this end.

Both use cases will also require a set of distributed protocols to support their efficient operation, namely membership protocols, data aggregation protocols, and potentially, data dissemination protocols (among others). In this context, the Yggdrasil runtime and associated protocols might simplify the task of developing (and potentially executing) the stack of distributed protocols to support applications and computations in the use cases.

The Yggdrasil framework will also allow to further explore the performance/overhead and precision (and the associated trade-off between these two aspects) of decentralized aggregation protocols in practice. While this will bring benefits to the use cases discussed above, this can also be useful to devise new aggregation strategies. Such strategies will potentially be useful in the context of the *monitoring systems of the Guifi.net Community* (provided by UPC), as aggregation is a good strategy to improve the operation of distributed monitoring mechanisms.

In particular, the monitoring schemes used in the Guifi.net could be improved by allowing relevant information to be gathered with lower communication overhead, and by aggregating this information directly at the edge, one could avoid the transmission of all monitoring information to a single point at the core of the network.

All of the use cases discussed above require data to be shared among edge devices, and also to be synchronized with centralized repositories. From this point of view, the contribution of Non-Uniform CRDTs can impact positively solutions designed to address the previous three use cases. Non-Uniform CRDTs allow to minimize the amount of data to be propagated among components while ensuring that computations still converge to a correct value (given the semantics of the application). Additionally, these CRDTs can be synchronized through pair-wise exchanges among replicas, in an uncoordinated fashion.

This synchronization strategy is well suited for all use cases discussed above, where we have a potentially large numbers of devices interacting with each other while collecting and processing data. This is reinforced by previous research and experimentation conducted in the Guifi.net context [4], where it was observed that gossip protocol-based communication provided better results than other alternatives for supporting distributed directories of services provided through the Guifi.net platform.

[4] http://wiki.clommunity-project.eu/pilots:distributed_announcement_and_discovery

6 Relationship with Results from other Work Packages

We now briefly discuss the relationship between the contributions presented and documented here and the work being conducted in the other Work Packages of the Lightkone project.

WP1: This work package is concerned with data protection and privacy, and it articulates with all other work packages including work package 5. At this point of the work however, we are still concerned with devising fundamental techniques and mechanisms to ease the development of edge-enabled applications. In the following months, we will start thinking about these aspects in the context of all presented contributions. We note however, that the Legion framework already presents mechanisms to ensure data privacy for users that leverage the framework.

WP2: The contributions and tools presented here will assist in supporting the design of novel solutions for addressing the use cases proposed previously in the context of this work package. We have discussed how we envision potential uses of the solutions documented here in this context in the previous section.

WP3: This work package focuses on designing solutions and methodologies to allow the inter-operation of components of edge-enabled (distributed) applications that lie at any point of the considered spectrum of edge scenarios, from the heavy edge to the light edge (where WP5 focus). In this context the secondary contribution, presented here, on Non-Uniform CRDTs, has been developed also in alignment with the goals of WP3. This contribution effectively offers a venue to devise a common data representation model that is useful for components more focused on the heavy edge as well as the light edge.

Additionally, the frameworks presented here (Yggdrasil, Legion, and Lasp) are all frameworks that assist in the design and execution of edge-enabled applications for multiple edge scenarios. All of these frameworks will evolve to adopt a common data representation model, as to allow them to inter-operate in the support of complex edge applications. This integration will be pursued in alignment with the work being conducted in WP3.

WP4: This work package is focused on devising semantics and programming abstractions for supporting edge applications, whose components might exist in any point of the heavy to light edge spectrum. Lasp, presented here, also exposes a programming interface that is fully detailed in D4.1.

An initial proposal for common semantics for edge applications and sub-systems has been proposed and derived from work conducted in the context of this work package. While the Lasp framework has been modeled in light of this common semantics, modeling the remaining support frameworks presented here is work that will be conducted in the future in close cooperation with the efforts of WP4.

WP6: While WP5 focuses on the light edge end of the spectrum, this work package focuses on the complementary heavy edge end of the spectrum. Naturally, we expect future edge-applications to be build of components that operate on multiple

points of this spectrum. The Legion framework already demonstrates how an edge-enabled application leverages on components that execute both at the core of the system and on user devices. Furthermore, Legion shows a way to integrate these two components, although it still lacks adapters for integration with AntidoteDB (reported in D6.1).

The secondary contribution presented in this document on enriching consistency guarantees through the use of a middleware running on the client side is also part of the work produced in the context of this work package. In fact, the goal of this contribution is to enrich the consistency exposed by (legacy) systems running on the heavy edge end of the spectrum.

WP7: The WP7 work package is focused on the evaluation of solutions and systems produced by the Lightkone consortium. While the efforts of this work package started recently, there are multiple contributions presented here that will be the target of evaluation by this work package.

In particular, the GRiSP board will be used to devise prototypes of embedded devices that can support novel edge-computing strategies being developed by the consortium. The frameworks presented here, will be evaluated in the context of WP7. Finally, we plan on leveraging the Yggdrasil framework to conduct a practical experimental evaluation of distributed data aggregation algorithms (among others) in the context of AdHoc wireless networks, as most existing work found in the literature has been experimentally validated and evaluated through simulation only.

7 Software

We now briefly present the four main software artifacts that are part of this deliverable. All software artifacts are publicly accessible through the Lightkone Work Package 5 public git repository at: https://github.com/LightKone/wp5-public.git

Yggdrasil Framework and Protocols: The current version of the Yggdrasil framework, and relevant membership and aggregation protocols are presented and discussed. The Yggdrasil framework was implemented in the C language, and targets linux-based execution environments. In particular, the framework (and associated protocols) has been extensively tested using a fleet of 24 Raspberry Pi 3.

Legion Framework: The Legion framework enables the development of web-based client applications that leverage on edge computing interactions among clients running the same application in a transparent way. We have evaluated the framework in multiple distributed contexts and browsers, including browsers running on smart mobile devices.

Lasp Framework: The Lasp framework allows to design and execute synchronization-free applications that resort to CRDTs as their data model. The Framework is highly scalable, being possible to run on thousands of machines simultaneously. Additionally, the framework showcases a set of support execution environments

tolls such as membership services and support for execution in container-based environments.

GRiSP Software: The GRiSP software allows programmers to develop Erlang programs that run in the GRiSP board (presented here). The software itself combines: *i*) the *GRiSP Erlang Runtime Library* that exports to Erlang operations directly related with the GRiSP board; and *ii*) the *rebar3_grisp* that provides the Rebar plug-in for the GRiSP project.

8 Final Remarks

This deliverable reports the main and secondary contributions of the Lightkone consortium on developing new technology and approaches to support a new generation of edge-enabled applications. In particular, this report covers the work conducted in the context of Work Package 5, that focuses on the fraction of the edge spectrum denoted as light edge. This entails devising adequate support for applications that are both available and correct while operating in a mostly independent way, away from the core of the system (i.e, components executing within data centers).

The report covers four main contributions and two secondary contributions. The main contributions focus on the support for edge-enabled applications, namely through a set of frameworks to support the development and execution of edge applications in concrete edge scenarios. Yggdrasil focuses on edge computations performed among resource constrained devices that are inter-connected through AdHoc networks. Legion focuses on enabling edge interactions for web applications by supporting direct browser-to-browser communication. Lasp focuses on edge scenarios closer to the core of the network, and allows to specify decentralized computations performed over CRDTs. Finally, GRiSP is a novel and general purpose embedded system and software component to allow the fast prototyping and testing of edge devices for scenarios related to Internet of Things and sensor networks.

The secondary contributions focus on supporting the inter-operation among components of edge-enabled applications. In particular, Non-Uniform CRDTs provide a first step in devising a data representation model for allowing the efficient exchange of data and replica synchronization among devices running on the light edge spectrum and also, between those devices and components running on the heavy edge spectrum. Our middleware solution, that allows to enrich the consistency guarantees provided by heavy edge components exposed to clients, allows to design clients that avoid to explicitly handle consistency anomalies exposed by legacy systems running in cloud infrastructures.

We also report on some of the relationships of the presented contributions with the use cases that have been identified by the Lightkone consortium previously. Additionally, we also discuss the relationship of these contributions with the work being conducted by the consortium in the context of other work packages.

As part of this deliverable we also present a set of software artifacts that are now publicly available through a git repository.

In the future, we shall continue developing the existing solutions to improve their support for edge-enabled applications in additional edge scenarios. We will also make an effort to invest in developing demonstrators of our ideas and results.

9 Papers and publications

Some of the results reported in this document have been published in the following publications. For convenience of the reader, we provide the full text of these publications in Appendix.

- Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In Proceedings of the 26th International Conference on World Wide Web (WWW '17). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, pp. 283-292. DOI: https://doi.org/10.1145/3038912.3052673

- Christopher S. Meiklejohn, Vitor Enes, Junghun Yoo, Carlos Baquero, Peter Van Roy, and Annette Bieniusa. 2017. Practical evaluation of the Lasp programming model at large scale: an experience report. In Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP '17). ACM, New York, NY, USA, pp. 109-114. DOI: https://doi.org/10.1145/3131851.3131862

- Gonçalo Cabrita and Nuno Preguiça. Non-uniform Replication. In Proceedings of the 21st International Conference on Principles of Distributed Systems (OPODIS'17). 18-20 December 2017, Lisboa, Portugal.

- F. Freitas, J. Leitão, N. Preguia and R. Rodrigues, Fine-Grained Consistency Upgrades for Online Services. 2017. IEEE 36th Symposium on Reliable Distributed Systems (SRDS), Hong Kong, 2017, pp. 1-10. DOI: https://doi.prg/10.1109/SRDS.2017.9

References

[1] Arduino - Products. https://www.arduino.cc/en/Main/Products.

[2] Craft and deploy bulletproof embedded software in Elixir Nerves Project. http://nerves-project.org/.

[3] libnl - Netlink Protocol Library Suite. https://www.infradead.org/ tgr/libnl/.

[4] Original implementation of pacman for browsers. http://github.com/daleharvey/pacman.

[5] Parse. http://parse.com.

[6] Raspberry Pi 3 Model B - Raspberry Pi. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[7] Raspberry Pi Zero. https://www.raspberrypi.org/products/raspberry-pi-zero/.

[8] WebRTC. http://www.webrtc.org/.

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

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

[11] 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.

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

[13] Deniz Altınbüken and Robbert Van Renesse. Ovid: A software-defined distributed systems framework. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[14] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *Proc. of VLDB Endow.*, 7(3), November 2013.

[15] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[16] Daniel Barbará-Millá and Hector Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal*, 3(3):325–353, July 1994.

[17] Mayank Bawa, Hector Garcia-Molina, Aristides Gionis, and Rajeev Motwani. Estimating aggregates on a peer-to-peer network. Technical Report 2003-24, Stanford InfoLab, April 2003.

[18] David Bermbach, Jorn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, IC2E '13, pages 114–123, Washington, DC, USA, 2013. IEEE Computer Society.

[19] Kenneth Birman and Robert Cooper. The isis project: Real experience with a fault tolerant programming system. In *Proceedings of the 4th Workshop on ACM SIGOPS European Workshop*, EW 4, pages 1–5, New York, NY, USA, 1990. ACM.

[20] Kenneth P Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM TOCS*, 17(2), 1999.

[21] Kenneth P. Birman and Robert V. Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[22] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[23] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[24] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 152–158, Feb 2004.

[25] Gonçalo Cabrita and Nuno Preguiça. Non-uniform Replication. In *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS 2017)*, 2017.

[26] N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. Emergent structure in unstructured epidemic multicast. In *Proc. of DSN'07*, UK, June 2007.

[27] Santiago Castiñeira and Annette Bieniusa. Collaborative offline web applications using conflict-free replicated data types. In *Proc. of PaPoC '15 Workshop*, 2015.

[28] Cisco. Cisco visual networking index: Global mobile data traffic forecast update. https://tinyurl.com/zzo6766, 2016.

[29] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. of VLDB Endow.*, 1(2), 2008.

[30] 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.

[31] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3), 2013.

[32] 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.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[34] 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.

[35] 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.

[36] T. Dillon, C. Wu, and E. Chang. Cloud computing: Issues and challenges. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 27–33, April 2010.

[37] Hoang T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611, 2013.

[38] EtherpadFoundation. Etherpad. etherpad.org.

[39] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[40] Facebook Inc. Continuing to build news feed for all types of connections. goo.gl/Q06CaL, December 2015.

[41] F. Freitas, J. Leitao, N. Preguia, and R. Rodrigues. Characterizing the Consistency of Online Services (Practical Experience Report). In *Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 638–645. IEEE, June 2016.

[42] S. Gajjar, N. Choksi, M. Sarkar, and K. Dasgupta. Comparative analysis of wireless sensor network motes. In *2014 International Conference on Signal Processing and Integrated Networks (SPIN)*, pages 426–431, Feb 2014.

[43] Ayalvadi Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Net. Group Comm.* 2001.

[44] Joseph Gentle. ShareJS API. github.com/share/ShareJS.

[45] Google Inc. Google Drive Realtime API. developers.google.com/google-apps/realtime/overview.

[46] G. Greenwald and E. MacAskill. Nsa prism program taps in to user data of apple, google and others. http://tinyurl.com/oea3g8t, 2013.

[47] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[48] Caroline Jay, Mashhuda Glencross, and Roger Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact.*, 14(2), August 2007.

[49] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. SOSP'95*, 1995.

[50] Joyent Inc. Node. js, 2014.

[51] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM TOCS*, 10(1), February 1992.

[52] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[53] J. Leitão. Topology management for unstructured overlay networks, sep 2012.

[54] J. Leitão, J. P. Marques, J. Pereira, and L. Rodrigues. X-bot: A protocol for resilient optimization of unstructured overlay networks. *IEEE TPDS*, 23(11), Nov 2012.

[55] João Leitão, José Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *Proc. of SRDS'07*. IEEE, 2007.

[56] João Leitão, José Pereira, and Luis Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Proc. of DSN'07*. IEEE, 2007.

[57] D. Lewis. icloud data breach: Hacking and celebrity photos. http://tinyurl.com/nohznmr, 2014.

[58] 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.

[59] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proc. 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[60] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

[61] 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.

[62] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN), RFC 5766. Technical report, IETF, April 2010.

[63] Christopher S. Meiklejohn, Vitor Enes, Junghun Yoo, Carlos Baquero, Peter Van Roy, and Annette Bieniusa. Practical evaluation of the lasp programming model at large scale: An experience report. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, pages 109–114, New York, NY, USA, 2017. ACM.

[64] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 707–710, Apr 2001.

[65] 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.

[66] Gabriel Montenegro, Christian Schumacher, and Nandakishore Kushalnagar. IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. RFC 4919, August 2007.

[67] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[68] David Navalho, Sérgio Duarte, and Nuno Preguiça. A Study of CRDTs That Do Computations. In *Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, 2015.

[69] P. C. Ng and S. C. Liew. Throughput analysis of ieee802.11 multi-hop ad hoc networks. *IEEE/ACM Transactions on Networking*, 15(2):309–322, April 2007.

[70] David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proc. UIST'95*, 1995.

[71] Patrick E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986.

[72] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V. Madhyastha, and Cristian Ungureanu. Simba: Tunable End-to-end Data Consistency for Mobile Apps. In *Proc. of EuroSys '15*, 2015.

[73] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proc. of NSDI'09*, 2009.

[74] C. M. Ramya, M. Shanmugaraj, and R. Prabakaran. Study on zigbee technology. In *2011 3rd International Conference on Electronics Computer Technology*, volume 6, pages 297–301, April 2011.

[75] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, October 2010.

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

[77] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine Partial Replication in Wide Area Networks. In *Proc. 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, 2010.

[78] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proc. of 13th SSS'11*, 2011.

[79] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[80] 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.

[81] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4), 2001.

[82] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of Comp. supported cooperative work*, 1998.

[83] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP'95*, 1995.

[84] Douglas B. Terry. *Replicated Data Management for Mobile Computing*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, 2008.

[85] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[86] 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.

[87] Marc Torrent-Moreno, Steven Corroy, Felix Schmidt-Eisenlohr, and Hannes Hartenstein. Ieee 802.11-based one-hop broadcast communications: Understanding transmission success and failure under different radio propagation environments. In *Proceedings of the 9th ACM International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '06, pages 68–77, New York, NY, USA, 2006. ACM.

[88] Philip W. Trinder. Comparing C++ and ERLANG for motorola telecoms software. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, page 51. ACM, 2006.

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

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

[91] Robbert van Renesse, Kenneth Birman, Dan Dumitriu, and Werner Vogels. Scalable management and data mining using astrolabe*. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pages 280–294, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

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

[93] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *J. of Net. & Sys. Manag.*, 13(2), 2005.

[94] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: Concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*, Mobidata '15, pages 37–42, New York, NY, USA, 2015. ACM.

[95] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[96] 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 *Proc. of Middleware'15*. ACM/IFIP/Usenix, December 2015.

A Pseudo-Code for the Aggregation Protocols

In this appendix the interested reader can find the pseudo-code for the aggregation algorithms discussed previously in § 3.1.

Algorithm 4 Broadcast Aggregation

```
1:  State:
2:    contributions                          ▷ set of contributions (process, value)
3:    reset                                  ▷ times the timer was reset
4:    threshold                              ▷ maximum number of reset times
5:    period                                 ▷ announce period
6:    aggValue                               ▷ aggregated value
7:    aggFunction                            ▷ aggregation function
8:
9:  Upon Init(t, period) do:
10:   reset ⟵ 0
11:   threshold ⟵ t
12:   period ⟵ period
13:
14: Upon RecvAggRequest(val, f) do:
15:   aggFunction ⟵ f
16:   aggValue ⟵ val
17:   contributions ⟵ (p, val)               ▷ where p is the process id
18:   Setup Timer Announce(period + RND)     ▷ where RND is a random small number
19:
20: Upon Announce() do:
21:   reset ⟵ reset + 1
22:   if (reset < threshold) then
23:     Trigger Send(SAGG_ANNOUNCEMENT, BROADCASTADDR, contributions)
24:     Setup Timer Announce(period + RND)
25:   else
26:     Trigger ReportValue(aggValue)
27:   end if
28:
29: Upon Receive(SAGG_ANNOUNCEMENT, c) do:
30:   newCon ⟵ c − contributions
31:   if newCon ≠ {} then
32:     reset ⟵ 0
33:     for all (p, v) ∈ newCon do
34:       aggValue ⟵ aggFunction(v, aggValue)
35:       contributions ⟵ contributions ∪ (p, v)
36:     end for
37:   end if
38:
```

Algorithm 5 Bloom Aggregation

State:
  contributions ▷ bloomfilter containing the process IDs that have already contributed to the aggregated value
  nContributions ▷ number of contributions in the bloomfilter
  myContribution ▷ the contribution of the process $(process, value)$
  neighbours ▷ the set of contributions of the neighbour processes $(process, value)$
  reset ▷ times the timer was reset
  threshold ▷ maximum number of reset times
  period ▷ announce period
  aggValue ▷ aggregated value
  aggFunction ▷ aggregation function

Upon Init(t, *period*) **do:**
  reset $\longleftarrow 0$
  threshold $\longleftarrow t$
  period $\longleftarrow$ *period*

Upon RecvAggRequest(val, f) **do:**
  aggFunction $\longleftarrow f$
  aggValue $\longleftarrow val$
  myContribution $\longleftarrow (p, val)$
  neighbours $\longleftarrow (p, val)$ ▷ where p is the process ID
  contributions $\longleftarrow \{\}$
  $contributions \longleftarrow contributions \cup p$
  Setup Timer Announce($period + RND$) ▷ where RND is a random small number

Upon Announce() **do:**
  reset $\longleftarrow reset + 1$
  if $(reset < threshold)$ **then**
    Trigger Send(BLOOM_ANNOUNCEMENT, *BROADCASTADDR*, *myContribution*, *aggValue*, *nContributions*, *contributions*)
    Setup Timer Announce($period + RND$)
  else
    Trigger ReportValue(*aggValue*)
  end if

Bloom Aggregation (continuation)

```
35: Upon Receive(BLOOM_ANNOUNCEMENT, (p,v), val, n, bloom) do:
36:   if contributions = bloom then
37:     return
38:   else if (contribution ∧ bloom) = {} then
39:     contribution ⟵ contribution ∪ bloom
40:     aggValue ⟵ aggFunction(aggValue, val)
41:     nContributions ⟵ nContributions + n
42:     if c ∉ neighbours then
43:       neighbours ⟵ neighbours ∪ (p,v)
44:     end if
45:     reset ⟵ 0
46:     return
47:   else
48:     nc ⟵ nContributions
49:     if p ∉ contributions then
50:       contributions ⟵ contributions ∪ p
51:       aggValue ⟵ aggFunction(v, aggValue)
52:       nContributions ⟵ nContributions + 1
53:       if (p,v) ∉ neighbours then
54:         neighbours ⟵ neighbours ∪ (p,v)
55:       end if
56:     end if
57:     for all (p,v) ∈ neighbours do
58:       if p ∉ bloom then
59:         bloom ⟵ bloom ∪ p
60:         val ⟵ aggFunction(v, val)
61:         n ⟵ n + 1
62:       end if
63:     end for
64:     if n > nContributions then
65:       aggValue ⟵ val
66:       nContributions ⟵ n
67:       contributions ⟵ bloom
68:     else if n = nContributions then
69:       if heuristic then          ▷ pick the information determined by an heuristic e.g, val > aggvalue
70:         aggValue ⟵ val
71:         contributions ⟵ bloom
72:       end if
73:     end if
74:     if nc < nContributions then
75:       reset ⟵ 0
76:     end if
77:   end if
78:
```

Algorithm 6 Single Tree Aggregation

```
1:  State:
2:    isRoot                                  ▷ whether we are the root node
3:    parent                                  ▷ parent node in the tree
4:    children                                ▷ set of children nodes
5:    toBeAccounted                           ▷ set of known nodes which have not answered to the node yet
6:    aggFunction                             ▷ aggregation function to perform
7:    aggValue                                ▷ current value of the node
8:
9:  Upon Init(neighbourhood, function) do:
10:   isRoot ⟵ False
11:   parent ⟵ ⊥
12:   children ⟵ {}
13:   toBeAccounted ⟵ neighbourhood
14:   aggFunction ⟵ function
15:   aggValue ⟵ ⊥
16:
17: Upon RecvAggRequest() do:
18:   isRoot ⟵ True
19:   aggValue ⟵ GetLocalValue()              ▷ retrieves the local value of the node
20:   Trigger Send(STREE_QUERY, BROADCASTADDR, aggFunction)
21:
22: Upon Receive(STREE_QUERY, s, function) do:
23:   if parent = ⊥ ∧ ¬isRoot then
24:     isRoot ⟵ False
25:     parent ⟵ s
26:     aggFunction ⟵ function
27:     aggValue ⟵ GetLocalValue()            ▷ retrieves the local value of the node
28:     Trigger Send(STREE_CHILD, s, YES)
29:     Trigger Send(STREE_QUERY, BROADCASTADDR, aggFunction)
30:   else
31:     Trigger Send(STREE_CHILD, s, NO)
32:   end if
33:
34: Upon Receive(STREE_CHILD, s, answer) do:
35:   if answer = YES then
36:     children ⟵ children ∪ s
37:   end if
38:   toBeAccounted ⟵ toBeAccounted \ s
39:   Trigger ReportAggregate()
40:
```

Single Tree Aggregation (continuation)

41: **Upon** Receive(STREE_VALUE, s, $receivedValue$) **do:**
42: aggValue $\longleftarrow$ $aggFunction(aggValue, receivedValue)$
43: $children \longleftarrow children \setminus s$
44: **Trigger** ReportAggregate()
45:
46: **Upon** ReportAggregate() **do:**
47: **if** $toBeAccounted \neq \{\} \vee children \neq \{\}$ **then return**
48: **end if**
49: **if** $isRoot$ **then**
50: **Trigger** ReportValue($aggValue$)
51: **else**
52: **Trigger** Send(STREE_VALUE, $parent$, $aggValue$)
53: **end if**
54:
55: **Upon** FaultDetected($node$) **do:**
56: $children \longleftarrow children \setminus node$
57: $toBeAccounted \longleftarrow toBeAccounted \setminus node$
58: **Trigger** ReportAggregate()
59:

B Relevant Publications

Legion: Enriching Internet Services with Peer-to-Peer Interactions*

Albert van der Linde[1] Pedro Fouto[1] João Leitão[1] Nuno Preguiça[1]
Santiago Castiñeira[2] Annette Bieniusa[2]
[1]NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa
[2]University of Kaiserslautern

ABSTRACT

Many web applications are built around direct interactions among users, from collaborative applications and social networks to multi-user games. Despite being user-centric, these applications are usually supported by services running on servers that mediate all interactions among clients. When users are in close vicinity of each other, relying on a centralized infrastructure for mediating user interactions leads to unnecessarily high latency while hampering fault-tolerance and scalability.

In this paper, we propose to extend user-centric Internet services with peer-to-peer interactions. We have designed a framework named Legion that enables client web applications to securely replicate data from servers, and synchronize these replicas directly among them. Legion allows for client-side modules, that we dub *adapters*, to leverage existing web platforms for storing data and to assist in Legion operation. Using these adapters, legacy applications accessing directly the web platforms can co-exist with new applications that use our framework, while accessing the same shared objects. Our experimental evaluation shows that, besides supporting direct client interactions, even when disconnected from the servers, Legion provides lower latency for update propagation with decreased network traffic for servers.

Keywords

Web Applications; Peer-to-Peer Systems; CRDTs; Frameworks

1. INTRODUCTION

A large number of web applications mediate interactions among users. Examples are plentiful, from collaborative applications, to social networks, and multi-user games. These applications manage a set of shared objects, the application state, and each user reads and writes on a subset of these objects. For example, in a collaborative text editor, users share the document being edited, while in a multi-user game the users access and modify a shared game state. In these cases, user experience is highly tied with how fast interactions among users occur.

These applications are typically implemented using a centralized infrastructure that maintains the shared state and mediates all interactions among users. This approach has several drawbacks. First, servers become a scalability bottleneck, as all interactions have to be managed by them. The work performed by servers has polynomial growth with the number of clients, as not only there are more clients producing contributions but also each contribution must be disseminated to a larger number of clients. Second, when servers become unavailable, clients become unable to interact, and in many cases, they cannot even access the application. Finally, the latency of interaction among nearby users is unnecessarily high since operations are always routed through servers. This might not be noticeable for applications with low interaction rates such as social networks. However, user experience in games and collaborative applications relies on interactive response times below 50ms [22].

One alternative to overcome these drawbacks is to leverage on direct interactions among clients, thus making the system less dependent on the centralized infrastructure. Besides avoiding the scalability bottleneck and availability issues of typical web application architectures, such an approach can also improve user experience by reducing the latency of interactions among clients. Additionally, it has the potential to lower the load imposed on centralized components, minimizing the infrastructure cost.

While there has been significant work in the design of peer-to-peer systems (*e.g.* [36, 28, 18, 9, 41, 27]), two main reasons prevented its adoption for improving web applications. First, web browsers restricted the ability to establish direct communication channels among clients. Recently, the Web Real Time Communication (WebRTC) initiative [5] has solved this limitation by enabling direct communication between browsers. Second, firewalls (and NAT boxes) restricted connectivity among client nodes. This problem can currently be circumvented by relying on widely available techniques, such as STUN and TURN [30]. Additionally, HTML5 makes it possible for these applications to locally store data that persists across sessions on the same browser. The combination of these techniques has created the opportunity for a new generation of web applications that can leverage peer-to-peer interactions.

In this work, we present Legion, a framework that exploits these new features for enriching web applications. Each client maintains a local data store with replicas of a subset of the shared application objects. We designed Legion to support web applications where groups of, at most, a few hundreds of users, collaborate by manipulating the same set of data objects. Legion adopts an

*This work was partially supported by FCT/MCTES: HYRAX project (CMUP-ERI/FIA/0048/2013); NOVA LINCS project (UID/CEC/04516/2013) and the European Union, through project Syncfree (grant agreement n°609551) and LightKone (grant agreement n°732505). Part of the computing resources used for this work were supported by an AWS in Education Research Grant.

WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3038912.3052673

eventual consistency model where each client can modify its local replica without coordination, while updates are propagated asynchronously to other replicas. To guarantee that all replicas converge to the same state despite concurrent updates, Legion relies on Conflict-free Replicated Data Types (CRDTs) [34]. CRDTs are replicated data types designed to provide eventual convergence without resorting to strong coordination.

Unlike systems [21, 32, 42, 4] that cache objects at the client, Legion clients can synchronize with the servers and directly among each other, using a peer-to-peer interaction model. To support these interactions, (subsets of) clients form overlay networks to propagate objects and updates among them. This induces low latency for propagating updates and objects between nearby clients.

Unlike uniform overlay networks [28, 18], Legion adopts a non-uniform design where a few selected nodes act as bridges between the client network and the servers that store data persistently. These *active nodes* upload updates executed by clients in the network and download new updates executed by clients that have not joined the overlay (including both legacy clients and clients unable to establish direct connections with other clients). This design reduces the load on the centralized component, which no longer needs to broadcast every update to all clients (nor track these clients).

While leveraging direct client interactions brings significant advantages, it also creates security challenges. We address these challenges by making it impossible for an unauthorized client to access objects or interfere with operations issued by authorized clients. Our design uses lightweight cryptography and builds on the access control mechanism of the central infrastructure to securely distribute keys among clients.

Client-side modules, *adapters*, allow Legion to, instead of using its own standalone servers, leverage existing web infrastructures for storing data and assist in several functions of the framework, including peer discovery, overlay management, and security management. As a showcase, we describe our adapters for Google Drive Realtime (GDriveRT), a Google service for supporting collaborative web applications similar to Google Docs [21]. The GDriveRT adapters allow Legion to: (i) store data in GDriveRT, while exposing an API and data model compatible with GDriveRT; (ii) support interaction between Legion-enriched clients accessing local object replica and legacy clients accessing the same GDriveRT objects directly; and (iii) resort to GDriveRT to assist in establishing initial peer-to-peer connections among clients.

Our evaluation shows that porting existing GDriveRT applications requires changing only a few lines of code (2 lines in the common case), allowing them to start benefiting from direct interactions among clients. We also show that the latency to propagate updates is much lower in Legion when compared with the use of a traditional centralized infrastructure, as in GDriveRT. Additionally, clients can continue to interact when the server becomes (temporarily) unreachable. Updates are stored locally and can be made durable by any active client when the server becomes available, either in the context of the same session or a future session. Since we avoid continuous access to the centralized infrastructure by all clients, the network traffic induced on the centralized component is lower, improving the scalability of the system. We also show that our security mechanisms have minimal overhead.

In summary, we present the design of Legion, a novel framework to enrich web applications through client side replication and (transparent) direct peer-to-peer interactions. To achieve this, and besides introducing the design of the Legion architecture, we make the following contributions:

- a data storage service for web clients, providing causal consistency and using CRDTs (§ 3.2);
- a topology-aware overlay-network core that uses WebRTC and promotes low-latency links between clients (§ 3.1.2);
- a lightweight security mechanism that protects privacy and integrity of data shared among clients (§ 3.3);
- a set of client adapters that integrate Legion with GDriveRT, storing data in the GDriveRT service, providing a seamless API and support for inter-operation with legacy clients (§ 4);
- the implementation (§ 5) and evaluation (§ 6) of a prototype that demonstrates the benefits of our approach in terms of latency for clients and reduced load on servers.

2. RELATED WORK

Our work has been influenced by prior research in multiple areas.

Internet services: Internet services often run in cloud infrastructures composed by multiple data centers, and rely on a geo-replicated storage system [15, 29, 7, 12, 13] to store application data. Some of these storage systems provide variants of weak consistency, such as eventual consistency [15] and causal consistency [29, 7], where different clients can update different replicas concurrently and without coordination. Similar to Google Drive Realtime, Legion adopts an eventual consistency model where updates to each object are applied in causal order.

Other storage systems adopt stronger consistency models, such as parallel snapshot isolation [35] and linearizability [13], where concurrent (conflicting) updates are not allowed without some form of coordination. Coordination among replicas for executing each update is prohibitively expensive for high throughput and large numbers of clients (manipulating the same set of data objects).

Replication at the clients: While many web applications are stateless, fetching data from servers whenever necessary, a number of applications cache data on the client for providing fast response times and support for disconnected operation. For example, Google Docs and Google Maps can be used in offline mode; Facebook also supports offline feed access [17].

Several systems that replicate data in client machines have been proposed in the past. In the context of mobile computing [38], systems such as Coda [25] and Rover [23] support disconnected operation relying on weak consistency models. Parse [4], SwiftCloud [42] and Simba [32] are recent systems that allow applications to access and modify data during periods of disconnection. While Parse provides only an eventual consistency model, SwiftCloud additionally supports highly available transactions [8] and enforces causality. Simba allows applications to select the level of observed consistency: eventual, causal, or serializability. In contrast to these systems, our works allows clients to synchronize directly with each other, thus reducing the latency of update propagation and allowing collaboration when disconnected from servers.

Bayou [39] and Cimbiosys [33] are systems where clients hold data replicas and that exploit decentralized synchronization strategies (either among clients [33] or servers [39]). Although our work shares some of the goals and design decisions with these systems, we focus on the integration with existing Internet services. This poses new challenges regarding the techniques that can be used to manage replicated data and the interaction with legacy clients, namely because most of these services can only act as storage layers (i.e., they do not support performing arbitrary computations).

Collaborative applications: Several applications and frameworks support collaboration across the Internet by maintaining replicas of shared data in client machines. Etherpad [16] allows clients to collaboratively edit documents. ShareJS [20] and Google Drive Realtime [21] are generic frameworks that manage data sharing among

multiple clients. All these systems use a centralized infrastructure to mediate interactions among clients and rely on operational transformation for guaranteeing eventual convergence of replicas [31, 37]. In contrast, our work relies on CRDTs [34] for guaranteeing eventual convergence while allowing clients to synchronize directly among them. Collab [11] uses browser plugins to allow clients in the same area network to replicate objects using peer-to-peer. Our work uses standard techniques for supporting collaboration over the Internet, requiring no installation by the end user, and allowing interaction with existing Internet services.

Peer-to-Peer systems: Extensive research on decentralized unstructured overlay networks [28, 41, 18] and gossip-based multicast protocols [9, 28, 10] have been produced in the past. Although our design for supporting peer-to-peer communication among clients builds on HyParView overlay network [28], it differs from this system in the way it promotes low latency links among clients and leverages the centralized infrastructure for handling faults efficiently.

3. SYSTEM DESIGN

Legion is a framework for data sharing and communication among web clients. It allows programmers to design web applications where clients access a set of shared objects replicated at the client machines. Web clients can synchronize local replicas directly with each other. For ensuring durability of the application data as well as to assist in other relevant aspects of the systems operation (discussed further ahead), Legion resorts to a set of centralized services. We designed Legion so that different Internet services (or a combination of Internet services and Legion's own support servers) can be employed. These services are accessed uniformly by Legion through a set of *adapters* with well defined interfaces.

By replicating objects in web clients and synchronizing in a peer-to-peer fashion, Legion reduces dependency and load on the centralized component (as the centralized component is no longer responsible to propagate updates to all clients), and minimizes latency to propagate updates (as they are distributed directly among clients). Furthermore, it allows clients (already running) to continue interacting when connectivity to servers is lost.

Figure 1 illustrates the client-side architecture of Legion with the main components and their dependencies/interactions:

Legion API: This layer exposes the API through which applications interact with our framework.

Communication Module: The communication module exposes two secure communication primitives: point-to-point and point-to-multipoint. Although these primitives are available to the application, we expect applications to interact using shared objects stored in the object store.

Object Store: This module maintains replicas of objects shared among clients, which are grouped in *containers* of related objects. These objects are encoded as CRDTs [34] from a pre-defined (and extensible) library including lists, maps, strings, among others. Web clients use the communication module to propagate and receive updates to keep replicas up-to-date.

Overlay Network Logic: This module establishes a logical network among clients that replicate some (shared) container. This network defines a topology that restricts interactions among clients, meaning that only overlay neighbors maintain (direct) WebRTC connections among them and exchange information directly.

Connection Manager: This module manages connections established by a client. To support direct interactions, clients maintain a set of WebRTC connections among them. (Some) Clients also maintain connections to the central component, as discussed below.

Legion uses two additional components that reside outside of the client domain:

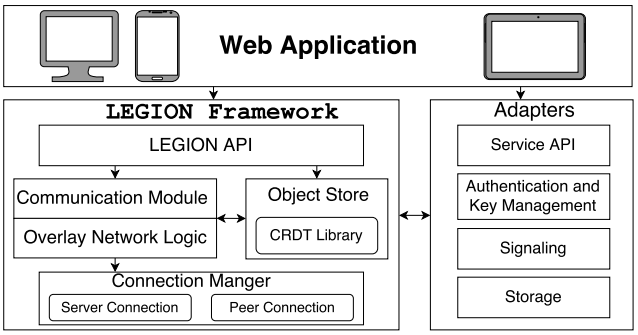


Figure 1: The Legion Architecture.

- one or more *centralized infrastructures* accessed through adapters for: (i) user authentication and key management (*authentication and key management* adapter); (ii) durability of the application state and support for interaction with legacy clients (*storage* adapter); (iii) exposing an API similar to the server API, thus simplifying porting applications to our system (*service API* adapter); (iv) assisting clients to initially join the system (*signaling* adapter);
- a set of STUN [30] servers, used to circumvent firewalls and NAT boxes when establishing connections among clients.

Our prototype includes adapters for GDriveRT and for a simple Node.js [24] server implemented by us. For STUN servers, our prototype relies on Google's public STUN servers.

In the remainder of this section we discuss in more detail the design of each of the modules that compose the Legion framework.

3.1 Communications

3.1.1 Communication Module

The communication module exposes an interface with point-to-point and multicast primitives, allowing a client to send a message to another client or to a group of clients. In Legion, each container has an associated multicast group that clients join when they start replicating an object from the container. Updates to objects in some container are propagated to all clients replicating the container.

Messages are propagated through the overlay network(s) provided by the *Overlay Network Logic* module. The multicast primitive is implemented using a push-gossip protocol (similar to the one presented in [28]).

Messages exchanged among clients are protected using a symmetric cryptographic algorithm, using a key (associated with each container) that is shared among all clients and obtained through the centralized component. Clients need to authenticate towards the centralized component to obtain this key, ensuring that only authorized (and authenticated) clients are able to observe and manipulate the objects of a container. We provide additional details about this mechanism further ahead.

3.1.2 Overlay Network Logic

Legion maintains an independent overlay for each container, defining the communication patterns among clients (*i.e.,* which clients communicate directly). The overlay is used to support the multicast group associated with that container.

Our overlay design is inspired by HyParView [28]. It has a random topology composed by symmetric links. Each client maintains a set of `K` neighbors (where `K` is a system parameter with values typically below 10 for medium scale systems as the ones we target in

this paper). Overlay links only change in reaction to external events (clients joining or leaving/failing).

In contrast to HyParView, we have designed our overlay to promote low latency links. As such, each client connects to $\mathtt{K}$ peers, with $\mathtt{K} = \mathtt{K}_n + \mathtt{K}_d$, where $\mathtt{K}_n$ denotes the number of *nearby* neighbors and $\mathtt{K}_d$ denotes the number of *distant* neighbors.

As shown by previous research [26], each client must maintain a small number of distant neighbors when biasing a random overlay topology to ensure global overlay connectivity and yield better dissemination latency while retaining the robustness of gossip-based broadcast mechanisms.

This requires clients to classify potential neighbors as being either nearby or distant. A common mechanism to determine whether a potential neighbor is nearby or distant is to measure the round-trip-time (RTT) to that node [26]. However, in Legion, since clients are typically running in browsers, it is impossible to efficiently measure round trip times between them, since a full WebRTC connection would have to be setup, which has non-negligible overhead due to the associated signaling protocol.

To circumvent this issue we rely on the following strategy that avoids clients to perform active measurements of RTT to other nodes. When a client starts, it measures its RTT to a set of W well-known web servers through the use of an HTTP HEAD request (the web servers employed in this context are given as a configuration parameter of the deployment). The obtained values are then encoded in an ordered tuple which is appended to the identifier of each client. These tuples are then used as coordinates in a virtual Cartesian space of W dimensions. This enables each client to compute a distance function between itself and any other client c given only the identifier of c.

3.1.3 Connection Manager

This module manages all communication channels used by Legion, namely *server connections* to the centralized infrastructure, and *peer connections* to other clients. We now briefly discuss the management of these connections.

Server Connections: A server connection offers a way for Legion clients to interact with the centralized infrastructure. We have defined an abstract connection that must be instantiated by the adapters that provide access to the centralized services. The connection for the Legion Node.js server uses web sockets. For the GDriveRT adapter, connections are established and authenticated for each container (document in GDriveRT). Independently of the employed centralized component, server connections are only kept open by *active clients*.

Peer Connections: A peer connection implements a direct WebRTC connection between two clients [1]. To create these connections, clients have to be able to exchange – out of band – some initial information concerning the type of connection that each endpoint aims to establish and their capacity to do so, which also includes information necessary to circumvent firewalls or NAT boxes using STUN/TURN servers. This initial exchange is known, in the context of WebRTC, as *signaling*.

Legion uses the centralized infrastructure for supporting the execution of the signaling protocol between a client joining the system and its initial overlay neighbors (that have to be active clients *i.e.,* with active server connections). After a client establishes its initial peer connections, it starts to use its overlay neighbors to find new peers. In this case, the signaling protocol required to establish these new peer connections is executed through the overlay network directly. If a client gets isolated and needs to rejoin the overlay, it relies again on the help of the centralized infrastructure. This greatly simplifies fault handling at the overlay management level.

The signaling adapter for GDriveRT stores information on a hidden document associated with the (main) document. Alternatively, clients can use the Legion native Node.js signaling server.

[1] Our experiments have shown that WebRTC connections can be established even among mobile devices using 3G/4G connectivity when devices use the same carrier.

3.2 Object Store

The object store maintains local replicas of shared objects, with related objects grouped in containers. Client applications interact by modifying these shared objects. Legion offers an API that enables an application to create and access objects.

CRDT library: Legion provides an extensible library of data types, which are internally encoded as CRDTs [34]. Objects are exposed to the application through (transparent) object handlers that hide the internal CRDT representation.

The CRDT library supports the following data types: Counters, Strings, Lists, Sets, and Maps. Our library uses Δ-based CRDTs [40], which are very flexible, allowing replicas to synchronize by using deltas with the effects of one or more operations, or the full state. This new type of delta-based CRDTs [6] is specially designed to allow efficient synchronization in epidemic settings, by avoiding, most of the times, a full state synchronization when two replicas connect for the first time. Each data type includes type-specific methods for querying and modifying its internal state, and generic methods to compute and integrate deltas (*i.e.,* differences).

Causal Propagation: This module uses the multicast primitive of the *Communication module* to propagate and receive deltas that encode modifications to the state of local replicas in a way that respects causal order (of operations encoded in these deltas). To achieve this, we use the following approach.

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

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

To prove that this approach respects causal order, we need to prove that when a delta d is received in a client c_r, all deltas that precede d in the causal order have already been received. This follows from the fact that if delta d has been received from client c_s and we know that client c_s sends deltas in causal order, then c_s has already sent all deltas that precede d in the causal order. By the same reason, the lists of deltas to propagate to other nodes in c_r also respect causal order after adding d to the end of their list. The formal proof for this property can be achieved by induction.

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

However, when two clients connect for the first time (or re-connect after a long period of disconnection), it might be impossible (or, at least, inefficient) to compute the adequate delta to send to its peer. In this case the two clients will synchronize their replicas by us-

ing the efficient initial synchronization mechanism supported by Δ-based CRDTs. In this case, if only a delta has been received, it is added to the list of deltas for propagation to other nodes. If it was necessary to synchronize using the full state, then the client needs to execute the same process to synchronize with other clients it is connected to.

3.3 Security Mechanisms

Allowing clients to replicate and synchronize among them a subset of the application state offers the possibility to improve latency and lower the load on central components. However, it also leads to concerns from the perspective of security, in particular regarding data privacy and integrity. In more detail, *privacy* might be compromised by allowing unauthorized users to circumvent the central system component to obtain copies of data objects from other clients; additionally, *integrity* can be compromised by having unauthorized users manipulate application state by propagating their operations to authorized clients.

We assume that an access control list is associated with each data container, and that clients either have full access to a container (being allowed to read and modify all data objects in the container) or no access at all. While more fine-grained access-control policies could easily be established, we find that this discussion is orthogonal to the main contributions of this paper. We also assume that the centralized infrastructure is trusted and provides an authentication mechanism that ensures that only authorized clients can observe and modify data in each container. Finally, we do not address situations where authorized clients perform malicious actions.

Considering these assumptions, Legion resorts to a simple but effective mechanism that operates as follows. The centralized infrastructure generates and maintains, for each container C, a persistent symmetric key K_C within that same container. Due to the authentication mechanism of the centralized infrastructure, only clients with access to a container C can obtain K_C. Every Legion client has to access the infrastructure upon bootstrap, which is required to exchange control information required to establish direct connections to other clients. During this process, clients also obtain the key K_C for the accessed container C.

K_C is used by all Legion clients to encrypt the contents of all messages exchanged directly among clients for container C. This ensures that only clients that have access to the corresponding container (and have authenticated themselves towards the centralized component) can observe the contents and operations issued over that container, addressing data privacy related challenges.

Whenever the access control list of a container is modified to remove some user, the associated symmetric key is invalidated, and a new key for that container is generated by the centralized infrastructure (we associate an increasing version number to each key associated with a given container).

To enable clients to detect when the key is updated in a timely fashion, the centralized component periodically generates a cryptographically signed message (using the asymmetric keys associated with the certificate of the server, used to support SSL connections) containing the current version of the key, and a nonce. This message is sent by the server to active peers, that disseminate the message (without being encrypted) throughout the overlay network.

If a client receives a message encrypted with a different key from the one it knows, either the client or its peer have an old key. When the client has an old key (with a version number smaller than the version number of the key used to encrypt the message), the client contacts the centralized infrastructure to obtain the new key. Otherwise, the issuer of the message has an old key and the client discards the received message and notifies the peer that it is using an old key. This will lead the sender of the message to connect to the central infrastructure (going again through authentication) to update the key before re-transmitting the message.

Note that clients that have lost their rights to access a container are unable to obtain the new key and hence, unable to modify the state of the application directly on the centralized component, send valid updates to their peers, or decrypt new updates.

While there might be a small increase in communication with the centralized infrastructure when a user's access is revoked (as a new key has to be generated and distributed), we believe that removing user permissions in collaborative web applications is not a frequent task. Furthermore, several access revocations can be compressed into a single update of the access control list (requiring only the generation and distribution of a single key).

4. ADAPTERS: GDriveRT

In this section we describe the adapters that can be used to integrate Legion with GDriveRT. In total (see Figure 1) we have implemented 4 distinct adapters with the following purposes: (i) a *storage* adapter enables Legion to outsource storage of application state and to (optionally) support GDriveRT legacy clients; (ii) a *signaling* adapter enables the use of GDriveRT to support signaling for establishing WebRTC connections; (iii) an *authentication and key management* adapter enables Legion to outsource to GDriveRT both user authentication and key management and distribution; and finally (iv) a *service API* adapter that exposes to client applications an interface similar to the GDriveRT API.

To simplify our prototyping, we have implemented these adapters as a single component, that enables the programmer to configure which adapters should be enabled (when an adapter is disabled, the functionality provided by it is delegated to the Legion Node.js server). In this section we discuss the most relevant aspects related with the design and implementation of these adapters which cover the specific challenges that a programmer faces when integrating Legion with an existing Web platform.

4.1 Data model

Our GDriveRT storage adapter supports the same data model as GDriveRT, in which collaboration among users is performed at the level of *documents*. A document contains a set of data objects and is mapped to a Legion container. Each object inside a document is mapped to an object of a similar type in Legion. The adapter transparently performs this mapping.

The associated service API adapter provides applications with an API similar to the GDriveRT API. The main functions of the API include a method to load a document that initializes the Legion framework. This method gives access to a handler for the document, which can be used by the application to read and modify the data objects included in the document state. As discussed, updates executed to the objects of a document's replica are delivered to other document replicas in causal order, i.e., Legion enforces causal consistency for operations over a document.

By exposing the same API of GDriveRT, this adapter enables any web application written in JavaScript that uses the GDriveRT API to be (easily) ported to Legion through the manipulation of a few lines of JavaScript code, namely: (i) adding an include statement to the script file with the code of Legion (and adapters) and (ii) replacing the function call to load a document by the equivalent function of the Legion GDriveRT service API adapter. With the handler for the loaded document, the application can use exactly the same function calls as in GDriveRT.

4.2 Legion functionality

Our Legion storage adapter can also leverage the GDriveRT infrastructure to: serve as a gateway between partitioned overlays that replicate the same GDriveRT document; and reliably store application state, i.e., documents and associated objects.

For serving as a gateway between partitioned overlays, for each document, the adapter maintains in GDriveRT the list of deltas of the document. As discussed before, in each overlay, a set of active clients is responsible to upload deltas executed by clients in the overlay and to download and disseminate new deltas throughout the overlay. If more than one client executes this process in each overlay, this does not affect correctness, as a delta received in a client is discarded if its timestamp is already reflected in the version vector of the replica. By the same reason that the list of deltas maintained by a client respects causal order, it follows that the list of deltas maintained in GDriveRT respects causal order. As a consequence, deltas downloaded from GDriveRT are also disseminated in an order that respects causality.

4.3 Support for Legacy Applications

While Legion allows web applications to explore peer-to-peer interactions using the Legion framework, it is also possible to allow legacy client applications to continue accessing data using the original GDriveRT interface by enabling a special option on our storage adapter (this support, as we show in § 6, incurs in some overhead).

When supporting legacy clients, for each data object, Legion keeps two versions: the version manipulated by Legion and the version manipulated by legacy applications. The key challenge is to keep both versions synchronized. This process is executed by a Legion client, as follows.

Applying operations executed in Legion clients to the GDriveRT object is a straightforward process that requires converting a delta stored in the list of executed deltas to the corresponding GDriveRT operations and executing them.

Applying operations executed in a GDriveRT object to the Legion object is slightly more complex because it is necessary to infer the executed operations. To this end, the client executing the synchronization process records the version number of the GDriveRT document in which the process is executed. In the next synchronization, the client infers the updates produced by legacy clients by comparing the current version of the document against the version after the last synchronization (using a diff algorithm). The updates are converted into a delta and added to the log of executed deltas, guaranteeing that the deltas are applied to Legion objects.

Both synchronization steps need to be executed by a single client to guarantee an exactly-once transfer of updates from one version to the other. We implement an election mechanism for selecting the client relying on a GDriveRT list. When no client is executing this process, a client willing to do it checks the version number of the document and the current size of the list, and then writes in the list the tuple $< id, n, t >$, with id being the client identifier, n the observed size of the list, and t the time until when the client will be executing the process (for periods in the order of seconds or minutes). The client then reads the following version of the document, which guarantees that its write has been propagated to GDriveRT servers. If the tuple the client has written is in position $n + 1$, the client is elected to execute the process. This is correct, as if two clients concurrently try to be elected, the tuple of only one will be in position $n + 1$ of the list in the new version of the document.

4.4 Security in GDriveRT

When using GDriveRT, we can leverage on the existing authentication mechanism of GDriveRT to perform access control. The security mechanism presented previously had to be slightly adapted as to ensure compatibility with the authentication and key management adapter due to the fact that GDriveRT only provides storage. GDriveRT exposes no computational capabilities, being therefore unable to generate symmetric keys, nor generate signed messages periodically to speed up the notification of clients of key changes.

To address these challenges we made the following modifications. First, when a new container C is created, the symmetric key K_C associated with that container is created by the first Legion client that accesses the container (after performing authentication).

Additionally, when a client removes a user's access to a container, it also generates a new key for that container. Using the GDriveRT authentication and key management adapter, active clients monitor the key, such that if the key is modified, they disseminate a notification through the overlay network, leading the remaining clients to access the centralized infrastructure to obtain it.

To deal with scenarios where users associated with all active peers have their access to a container revoked, every client periodically contacts the centralized component to verify that the known key is still valid. This step can be performed infrequently because, as soon as a single client becomes aware of a new key, the knowledge that a new key exists is epidemically propagated throughout the overlay network, as that client will start to use the new key to encrypt all messages exchanged with its neighbors, leading them to fetch the new key from the centralized infrastructure.

5. IMPLEMENTATION DETAILS

We now provide a few implementation details of our prototype. The code is available at: `github.com/albertlinde/Legion`.

Overlay networks: To achieve the threshold of `K` neighbors we do the following. Upon joining the system, a client resorts to the centralized component (either the Legion server or another web service accessed through a specialized adapter) to obtain the identifiers of nodes that currently have an open connection to the centralized infrastructure. Using this information, the client establishes a connection to a nearby neighbor and another to a distant neighbor (for these connections the centralized infrastructure is leveraged to perform the WebRTC signaling protocol). Random walks over the existing overlay are then used by the new client to find other nearby and distant neighbors to fill its neighborhood.

To classify peers as being either nearby or distant we resort to the previously described scheme, where we use 4 distinct sites, which are endpoints of Amazon EC2 Web API (scattered throughout 4 different data centers). While different distance functions can be employed over the virtual coordinates associated with each client, in our prototype we use a function that categorizes a client to be *distant* if the difference between at least two coordinates in the 4D virtual space are equal or above 70, and *nearby* otherwise (we have experimentally asserted that this strategy yields adequate results).

Selection of Active Clients: In our design, we use a small subset of clients (that we dub *active clients*) to upload and download updates over objects to and from the centralized infrastructure and to monitor the cryptographic key associated with each container. To select these clients we use a bully algorithm [19] where initially all clients act as an active client, and periodically, every $\mathtt{T}ms$, sends to its nearby overlay neighbors a message containing its unique identifier – in our experiments we set $\mathtt{T} = 7000\ ms$. Whenever a client receives a notification from a neighbor whose identifier is lower than its own, it switches its own state to become a *passive client*, and stops disseminating periodic announcements. To address the departure or failure of active clients, if a *passive client* does not receive an announcement for more than $3 \times \mathtt{T}\ ms$, it switches its

own state back to become an *active client* (the factor of 3 is used to avoid triggering this process unnecessarily).

Passive clients disable their connection to the centralized infrastructure. The result of executing this algorithm is that only a small subset of (non-neighboring) clients remain *active clients*.

Security Mechanisms: For the symmetric cryptography algorithm, we used AES operating in block cipher mode, using a key of 128 bits. We use RSA, configured with a key of 2048 bits, for generating and verifying the signature of the messages issued by our Node.js server. Our implementation resorts to the Forge [1] JavaScript library to implement all cryptographic operations.

STUN Service: In our prototype we have resorted to publicly available Google STUN servers. However, this is a configurable aspect in our prototype, and can easily be modified to use privately owned and managed servers if an application operator desires.

Implementation complexity: We have used `Count Lines of Code` [14] and verified that the code for our GDriveRT adapters has 1.768 JavaScript LOC, while the whole implementation of Legion (including the simple server for materializing the centralized component) has 4.639 JavaScript LOC.

6. EVALUATION

This section presents an evaluation of Legion with an emphasis on the operation of Legion when using the adapters to inter-operate with the GDriveRT infrastructure (except if specifically stated in our experiments, we ran Legion with all GDriveRT adapters enabled and with support for legacy clients disabled). Our evaluation mainly focus on two complementary aspects. We start with an analysis of our experience in adapting existing GDriveRT applications to leverage Legion. Then, we present an experimental evaluation of our prototype, comparing it to the centralized infrastructure of GDriveRT regarding the following practical aspects: (i) What is the impact on update propagation latency? (ii) What is the impact on application performance? (iii) How does the system behave when the central server becomes (temporarily) unavailable? (iv) What is the impact of using Legion in terms of load imposed on the central component and on individual clients? (v) What is the overhead for supporting seamless integration with legacy clients?

6.1 Designing Applications

In this section, we describe a set of web applications that we have ported to Legion using the GDriveRT adapters.

Google Drive Realtime Playground: The Google Drive Realtime Playground [2] is a web application showcasing all data-types supported by GDriveRT. We ported this application to Legion by changing only 2 lines in the source code (see § 4).

Multi-user Pacman: We adapted a JavaScript version of the popular arcade game Pacman [3] to operate under the GDriveRT API with a multi-player mode. We also added support for multiple passive observers that can watch a game in real time. In our adaptation up to 5 players can play at the same time, one player controlling Pacman (the hero) and the remaining controlling each of the four Ghosts (enemies).

The Pacman client is responsible for computing, and updating the adequate data structures, with the *official* position of each entity. Clients that control Ghosts only manipulate the information regarding the direction in which they are moving. If no player controls a Ghost, its direction is determined by the the original game's AI, running in the client controlling Pacman.

In this game, we employed the following data types provided by the GDriveRT API: (i) a map with 5 entries, one for Pacman and the remaining for each Ghost, where each entry contains the identifier (ID) of the player controlling the character (each user generates its own random ID); (ii) a list of events, that is used as a log for relevant game events, which include players joining/leaving the game, a Ghost being eaten, Pacman being captured, etc. (iii) a list representing the game map, used to maintain a synchronized view of the map between all players. This list is modified, for instance, whenever a *pill* is eaten by Pacman; (iv) a map with 2 entries, one representing the width and the other the height of the map. This information is used to interpret the list that is used to encode the map; (v) a map with 2 entries, one used to represent the state of the game (paused, playing, finished) and the other used to store the previous state (used to find out which state to restore to when taking the game out of pause); finally, (vi) 5 maps, one for each playable character, with the information about each of these entities, for maintaining a synchronized view of their positions (this is only altered when the corresponding entity changes direction, not at every step), directions, and if a ghost is in a vulnerable state.

Along with extending and porting this application to use the GDriveRT API, we also implemented the same game (with all functionality) using Node.js as a centralized server for the game through which the clients connect using web-sockets (this implementation does not leverage Legion). This enables us to investigate the effort in implementing such an interactive application using both alternatives. The Node.js implementation of the game is approximately 2.200 LOC for the client code, and 100 LOC for the server. In contrast, the implementation leveraging the GDriveRT API has approximately 1.620 LOC for the client code, and 40 lines of code for the server side (used to run multiple games in parallel). This shows that an API such as the one provided by GDriveRT and Legion simplifies the task of designing such interactive web applications.

Creating the Legion version (using the GDriveRT adapters) required to change only two lines of code of the GDriveRT version (as described before). From a user perspective, the Legion version runs much smoother, which is also shown by our evaluation presented further ahead.

Spreadsheet: We have also explored an additional application: a collaborative spreadsheet editor. Each spreadsheet represents a grid of uniquely identifiable rows and columns, whose intersection is represented by an editable cell. Each cell can hold numbers, text, or formulas that can be edited by different users.

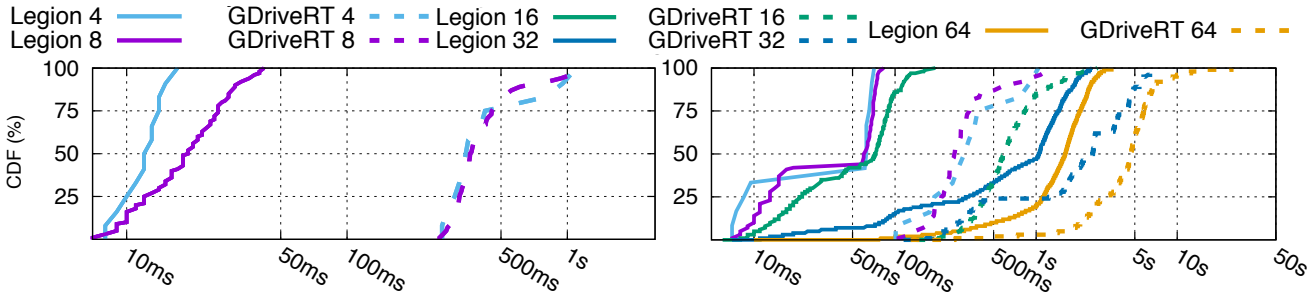
A prototype of the spreadsheet web application was built using AngularJS and supporting online collaboration through GDriveRT. The spreadsheet cells were modeled using a GDriveRT map. Each cell was stored in the map using its unique identifier (row-column) as key. Porting this application to the Legion API only required the change of 2 lines of code (as discussed previously).

Discussion: Our experience with porting these applications to leverage Legion shows that doing so is simple, as the programmer can easily use our GDriveRT adapters. Furthermore, this shows that carefully designing our framework to expose (through adapters) APIs that are similar to existing Web infrastructures is paramount to promote easy adoption of our solutions.

6.2 Experimental evaluation

In our experimental evaluation, we compare Legion, with and without the use of adapters, against GDriveRT, as a representative system that uses a traditional centralized infrastructure.

In our experiments, we have deployed clients in two Amazon EC2 datacenters, located at North Virginia (us-east-1) and Oregon (us-west-2). In each DC, we run clients in 8 m3.xlarge virtual machines with 4 vCPUs of computational power and 15GB of RAM. Unless stated otherwise, clients are equally distributed over both DCs. The average round-trip time measured between two machines in the same DC is 0.3 ms and 83 ms across DCs.



(a) All clients within the same datacenter

(b) Clients distributed over 2 Datacenters

Figure 2: Latency for the propagation of updates.

Latency: To measure the latency experienced by clients for observing updates, we conduct the following experiment. Each client inserts in a shared map a key-value pair consisting of his identifier and a timestamp. When a client observes an update on this map, it adds to a second map, as a reply, another pair concatenating the originating identifier and the replier's identifier as the key, and as value an additional timestamp. When a client observes a reply to his message, it computes the round-trip time for that reply, with latency being estimated as half of that time. All clients start by writing to the first map at approximately the same time and reply to all identifiers added by other clients. Thus, this simulates a system where the load grows quadratically with the number of clients.

Figure 2 presents the latency observed by all clients for both Legion and GDriveRT. The results show that latency using Legion is much lower than using GDriveRT for any number of clients. The main reason for this is that the propagation of updates does not have to incur a round-trip to the central infrastructure in Legion. Furthermore, for 64 clients, the 95^{th} percentile for GDriveRT is almost an order of magnitude above Legion, suggesting that Legion's peer-to-peer architecture is better suited to handle higher loads than the centralized architecture of GDriveRT.

Multi-Player Pacman Performance: We now show the impact of Legion on the performance of applications in the context of our Multi-player Pacman game.

To that end we conducted an experiment with volunteers, where we had five users playing Pacman (one player controlling Pacman, and four players for each of the ghosts). This experiment was conducted using five machines, in a local area network. Machines were running Ubuntu and clients executed in Firefox.

We focus our experiments in measuring the displacement of entities in relation to their official position. As explained before, each client updates an object with the direction of its movement. The Pacman client computes and updates the official position of each entity periodically. Each client independently updates its interface based on the known direction of movement and the latest official positions. Displacement captures the difference between the position computed by a client and the received official position upon receiving an update. When displacement has large values, users see entities jumping on the game map. In particular we measure: (i) the displacement of Pacman in relation to an eaten pill when an update reporting the pill being eaten is received by a client controlling a Ghost; and (ii) the displacement of Pacman and Ghosts when a client controlling a ghost receives an update for a position. Figure 3 reports the obtained results where the displacement is measured in tiles (the square unit that forms the interface). The board size of Pacman was 19×22 tiles featuring approximately, 59 turning points. Pacman and Ghosts move at approximately 3.33 tiles per second.

Figure 3(a) shows that when using Legion the interface is much more synchronized in relation to the real state of the system, showing that Pacman is visible by other players much closer to the eaten pill than when using the GDriveRT version of the game. Figure 3(b) reinforces these results showing that when using Legion the displacement of entities in the game interface is significantly lower when compared with the game version that only uses GDriveRT, which is unable to send updates to all clients at an adequate rate.

Effect of disconnection: We study the effect of disconnection by measuring the fraction of updates received by a client. In the presented results, clients share a map object, and each client executes one update per second to the map (similar behavior was observed with other supported objects). We simulate a disconnection from the Google servers, by blocking all traffic to the Google domain using *iptables*, 80 seconds after the experiment starts. The disconnection lasts for 100 seconds, after which rules in iptables are removed so that connections can again be re-established.

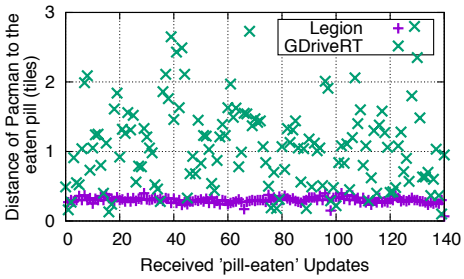
Figure 4 shows, at each moment, the average fraction of updates observed by clients since the start of the experiment (computed by dividing the average number of updates received by the total number of updates executed). As expected, the results show that during the disconnection period, GDriveRT clients no longer receive new updates, as the fraction of updates received decreases over time. When connectivity is re-established, GDriveRT is able to recover. With Legion, as updates are propagated in a peer-to-peer fashion, the fraction of updates received is always close to 100%.

We note however, that while servers remain inaccessible, new clients cannot join the system. However, when leveraging Legion, clients that are active when the server becomes unavailable can continue operating regularly without noticing the server unavailability.

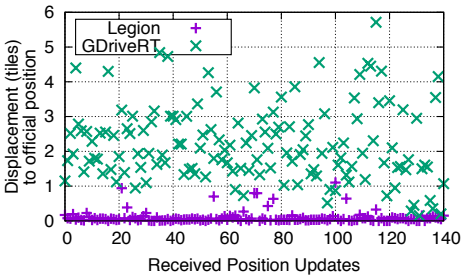
Network load: We now study the network load induced by our approach. To this end, we run experiments where 16 clients share a map object. Each client executes one update per second. The workload is as follows: 20% of updates insert a new key-value pair and 80% replace the value of an existing key selected randomly. The keys and values are strings of respectively 8 and 16 characters. We measure the network traffic by using *iptraf*, an IP network monitor.

In these experiments, we used the following configurations: *Legion w/ Node.js*: that uses our Legion server as backend. *Legion w/ GDriveRT*: that uses GDriveRT documents as backend. *GDriveRT*: that uses the original GDriveRT document as backend.

Figure 5(a) shows the total network load of the setup process, which entails making the necessary connections to the infrastructure and peer-to-peer connections. The incurred load using our own backend server is due to clients requiring to use this component to



(a) Pacman displacement to eaten pills



(b) Pacman and Ghosts Displacement

Figure 3: Muti-User Pacman Performance assessment.

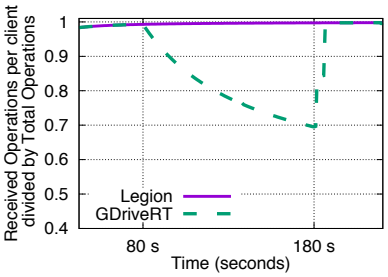
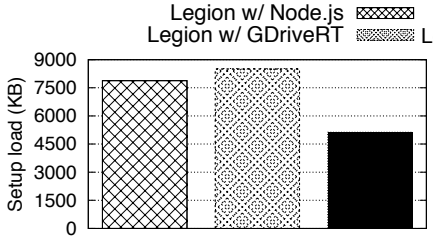
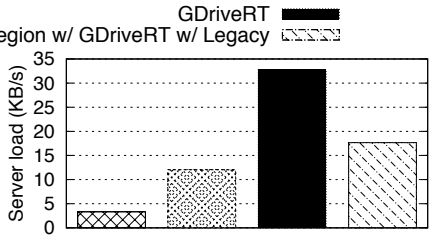


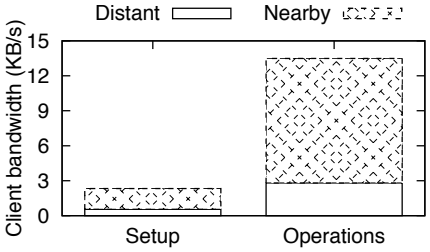
Figure 4: Effect of disconnection



(a) Server load during setup



(b) Server load during operations



(c) Client-to-client load

Figure 5: Network load.

connect to each other initially (WebRTC signaling). Legion using GDriveRT as backend has a slightly higher cost due to the overhead of performing signaling through the infrastructure, which is less efficient. In both cases only few clients obtain the initial object and propagate to other clients. Finally, in GDriveRT all clients download the shared data from the infrastructure.

Figure 5(b) shows the network load of the server without considering the initial setup load (computed by adding the traffic of all clients to and from the centralized infrastructure) for all competing alternatives. Results show that the load imposed over the centralized component is much lower when using Legion with GDriveRT as backend than when using only GDriveRT. This is expected, as only a few clients (active clients) interact with the GDriveRT infrastructure, being most interactions propagated directly between clients. Interestingly, the use of our server leads to an even lower load on the centralized component, this happens not only because the signaling mechanism used to establish new WebRTC connections among clients and the process for replica synchronization with the server is more efficient, but also because the data representation used by our backend is significantly more compressed. We run an additional configuration, (*Legion with GDriveRT w/ Legacy*) that uses GDriveRT documents as backend and synchronizes with the original document every 5 seconds. Supporting legacy clients (i.e., synchronizing with the original document) incurs a non negligible overhead. This happens because the mechanism used requires a large number of accesses to the centralized infrastructure as to infer which operations should be carried from legacy clients to the Legion clients and vice versa. However, even with support for legacy clients enabled, Legion induces lower load on the centralized component when compared with GDriveRT.

Figure 5(c) reports the average peer-to-peer communication traffic for each client during the setup of WebRTC connections (*Setup*) and while clients issue and propagate operations (*Operations*). The results show that the traffic of each client is larger than the traffic of each client with the server in GDriveRT (which can be approximated by dividing the server load – in Figure 5(b) – by the number of clients). This happens because our dissemination strategy has inherent redundancy, whereas in GDriveRT there are no redundant transmissions between each client and the centralized infrastructure. However, an average under 14KBps does not represent a huge fraction of available bandwidth nowadays. Furthermore the use of our location aware overlay leads to a network usage pattern where the amount of data sent to distant nodes is significantly lower that that sent to nearby nodes.

7. FINAL REMARKS

In this paper we presented the design of Legion, a framework that allows the development of web applications with seamless support for replication at the client machine leveraging peer-to-peer interactions to propagate operations among clients. Furthermore, we presented the design of adapters that enable Legion to leverage GDriveRT for (potentially) multiple purposes namely, storage backend; WebRTC signaling; authentication and key management; exposing an API akin to that of GDriveRT; and finally, an optional mechanism to support the co-existence of legacy clients.

The evaluation of our prototype shows that latency for update propagation is much lower using Legion when compared with the use of GDriveRT. Furthermore, load to the centralized infrastructure is greatly reduced by leveraging peer-to-peer interactions. Finally, we show that clients are able to interact while servers are temporarily unavailable. As future work, we plan to study how to support applications with thousands of simultaneous users, and to design and implement adapters to integrate Legion with storage services such as Cassandra, Redis, and Antidote.

8. REFERENCES

[1] Forge. github.com/digitalbazaar/forge.

[2] Google Drive Realtime Playground. github.com/googledrive/realtime-playground.

[3] Original implementation of pacman for browsers. github.com/daleharvey/pacman.

[4] Parse. parse.com.

[5] WebRTC. www.webrtc.org/.

[6] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. In *Proc. of NETYS'15*, Morocco, 2015.

[7] S. Almeida, J. Leitão, and L. Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. of EuroSys'13*, 2013.

[8] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *Proc. of VLDB Endow.*, 7(3), Nov. 2013.

[9] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM TOCS*, 17(2), 1999.

[10] N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. Emergent structure in unstructured epidemic multicast. In *Proc. of DSN'07*, UK, June 2007.

[11] S. Castiñeira and A. Bieniusa. Collaborative offline web applications using conflict-free replicated data types. In *Proc. of PaPoC '15 Workshop*, 2015.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. of VLDB Endow.*, 1(2), 2008.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3), 2013.

[14] A. Danial. Cloc–count lines of code. *Open source*, 2009.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6), 2007.

[16] EtherpadFoundation. Etherpad. etherpad.org.

[17] Facebook Inc. Continuing to build news feed for all types of connections. goo.gl/Q06CaL, Dec. 2015.

[18] A. Ganesh, A.-M. Kermarrec, and L. Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Net. Group Comm.* 2001.

[19] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Tran. on Comp.*, C-31(1), Jan 1982.

[20] J. Gentle. ShareJS API. github.com/share/ShareJS.

[21] Google Inc. Google Drive Realtime API. developers.google.com/google-apps/realtime/overview.

[22] C. Jay, M. Glencross, and R. Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Trans. Comput.-Hum. Interact.*, 14(2), Aug. 2007.

[23] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. SOSP'95*, 1995.

[24] Joyent Inc. Node. js, 2014.

[25] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM TOCS*, 10(1), Feb. 1992.

[26] J. Leitão, J. P. Marques, J. Pereira, and L. Rodrigues. X-bot: A protocol for resilient optimization of unstructured overlay networks. *IEEE TPDS*, 23(11), Nov 2012.

[27] J. Leitão, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *Proc. of SRDS'07*. IEEE, 2007.

[28] J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *Proc. of DSN'07*. IEEE, 2007.

[29] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. of SOSP'11*, 2011.

[30] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN), RFC 5766. Technical report, IETF, Apr. 2010.

[31] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proc. UIST'95*, 1995.

[32] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu. Simba: Tunable End-to-end Data Consistency for Mobile Apps. In *Proc. of EuroSys '15*, 2015.

[33] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proc. of NSDI'09*, 2009.

[34] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *Proc. of 13th SSS'11*, 2011.

[35] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. of SOSP'11*, 2011.

[36] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4), 2001.

[37] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of Comp. supported cooperative work*, 1998.

[38] D. B. Terry. *Replicated Data Management for Mobile Computing*. Synthesis Lectures on Mobile and Pervasive Computing. Morgan & Claypool Publishers, 2008.

[39] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of SOSP'95*, 1995.

[40] A. van der Linde, J. Leitão, and N. Preguiça. Δ-crdts: Making δ-crdts delta-based. In *Proc. of the PaPoC'16 Workshop*, United Kingdom, 2016. ACM.

[41] S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *J. of Net. & Sys. Manag.*, 13(2), 2005.

[42] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. In *Proc. of Middleware'15*. ACM/IFIP/Usenix, Dec. 2015.

Practical Evaluation of the Lasp Programming Model at Large Scale

An Experience Report

Christopher S. Meiklejohn*
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Vitor Enes
HASLab / INESC TEC
Universidade do Minho
Braga, Portugal

Junghun Yoo
University of Oxford
Oxford, United Kingdom

Carlos Baquero†
HASLab / INESC TEC
Universidade do Minho
Braga, Portugal

Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Annette Bieniusa
Technische Universität Kaiserslautern
Kaiserslautern, Germany

ABSTRACT

Programming models for building large-scale distributed applications assist the developer in reasoning about consistency and distribution. However, many of the programming models for weak consistency, which promise the largest scalability gains, have little in the way of evaluation to demonstrate the promised scalability. We present an experience report on the implementation and large-scale evaluation of one of these models, Lasp, originally presented at PPDP '15, which provides a declarative, functional programming style for distributed applications. We demonstrate the scalability of Lasp's prototype runtime implementation up to 1024 nodes in the Amazon cloud computing environment. It achieves high scalability by uniquely combining hybrid gossip with a programming model based on convergent computation. We report on the engineering challenges of this implementation and its evaluation, specifically related to operating research prototypes in a production cloud environment.

*Partially funded by the SyncFree Project in the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 609551, by the LightKone Project in the European Union Horizon 2020 Framework Programme for Research and Innovation (H2020/2014-2020), under grant agreement n° 732505, and by the Erasmus Mundus Doctorate Programme under grant agreement n° 2012-0030.

†Work was partially supported by SMILES within project "TEC4Growth – Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF); EU FP7 SyncFree project (609551), and EU H2020 LightKone project (732505).

PPDP'17, October 9–11, 2017, Namur, Belgium

ACM ISBN 978-1-4503-5291-8/17/10…$15.00
https://doi.org/10.1145/3131851.3131862

ACM Reference Format:
Christopher S. Meiklejohn, Vitor Enes, Junghun Yoo, Carlos Baquero, Peter Van Roy, and Annette Bieniusa. 2017. Practical Evaluation of the Lasp Programming Model at Large Scale. In *Proceedings of PPDP'17, Namur, Belgium, October 9–11, 2017,* 6 pages.
https://doi.org/10.1145/3131851.3131862

1 INTRODUCTION

Once a specialized field for applications that required large data sets, large-scale distributed applications have become commonplace in our globalized society. Regardless of whether you are developing a rich-web application or a native mobile application, managing distributed data is challenging. For simplicity, developers today typically resort to using a single database that provides a form of strong[1] consistency. In essence, the database serves as shared memory for the clients in the system.

A single database is an obvious bottleneck as it introduces a serialization point for all operations; this restricts the possible throughput of the system. As developers strive to provide a near-native experience where operations appear to happen immediately, and since not all clients can be geographically located close to the database, application performance can suffer as users move farther from the database; or worse, when clients can't communicate with the database at all because they are offline. To provide good user experience, including high availability and low latency, developers are forced to integrate replication in the system design.

Systems that favor weak consistency scale better: data items can be locally replicated, locally mutated by the application, and their state can be disseminated asynchronously, outside of the critical path. Weak consistency allows applications to continue to operate while offline. While these systems provide for high scalability and high performance, programming with weak consistency can be a challenge for the application developer as updates to data items have no guarantee on update visibility or update order. Concurrency poses an additional problem, as updates happening concurrently at different replicas may be conflicting.

Numerous systems and programming models[2, 3, 6, 8, 11, 14, 17, 18] have been proposed for working with weak consistency,

[1] For instance, linearizability, where a value follows the real-time order of updates.

however few have seen adoption. Many of the systems have sound theoretical foundations, but few perform evaluations at scale to demonstrate the benefits in practice. We believe that the lack of these results comes from the difficulty in the required infrastructure for large-scale experiments, and the challenges in engineering an implementation of a theoretical model using existing software languages and libraries.

In this paper, we discuss the practical issues encountered when evaluating one of these programming models, Lasp [4, 5], originally presented at PPDP '15. Lasp is designed using a holistic approach where the programming model was co-designed with its runtime system to ensure scalability. We examine the challenges of engineering an implementation capable of scaling to a large number of nodes running in a public cloud environment, using a real world application scenario. Further, we report on the engineering challenges of demonstrating the scalability of the Lasp model. Our experience report substantiates that empirically validating scalability is non-trivial, regardless of the programming model.

2 ADVERTISEMENT COUNTER

Lasp was invented to ease the development of distributed applications with weak consistency. The advertisement counter scenario from Rovio Entertainment, creator of Angry Birds, is an ideal fit for Lasp. This application counts the total number of times each advertisement is displayed on all client mobile phones, up to a given threshold for each. The application has the following properties:

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

As part of the large-scale evaluation done in the SyncFree project, and following the personal curiosity of the developers, we decided to invest resources in using industrial-strength engineering techniques to evaluate the scalability of this application running in a real world production cloud environment.

2.1 Lasp

Lasp [11] is a programming model that allows developers to write applications with Conflict-Free Replicated Data Types (CRDTs) [1, 16]. CRDTs are abstract data types, designed for use in concurrent and distributed programming, that have a binary merge operation to join any two replicas of a single CRDT. Under concurrent modification without coordination, different replicas of a single CRDT may diverge; the merge operation supports value convergence by ensuring that given enough communication, all replicas, without coordination, will converge to a single deterministic value regardless of the order that data is received and merged.

Historically, before CRDTs were introduced, ad-hoc merge functions were used, often with few formal guarantees. Later, after their development, programmers who wanted to use CRDTs in their applications would have two choices: either, using a single CRDT from existing literature to store application state, fitting their problem to an existing data structure; or, building a custom CRDT that fits their application domain, which requires to ensure that the merge operation is both deterministic and convergent.

Lasp improves this choice in two ways:

- **Composition.** Lasp provides set-theoretic and functional combinators for composing CRDTs into larger CRDTs.
- **Monotonic conditional.** Lasp introduces a conditional operation that allows the execution of application logic based on monotonic conditions[2] on CRDTs.

These two concepts allow Lasp applications to be both transparently and arbitrarily distributed across a set of nodes without altering application behavior. For brevity, the reader is referred to [11] for a full treatment of the Lasp semantics.

The advertisement counter uses two data structures from Lasp: the Add-Wins Set CRDT[3], where elements can be arbitrarily removed and inserted without coordination and under concurrent add and remove operations the add will 'win'; and the Grow-Only Counter CRDT, which models a counter that only increments.

2.2 Overview

The design of the advertisement counter is roughly broken into three components.

- **Initialization.** When the advertisement counter application is first initialized, we first create Grow-Only Counters for each unique advertisement we want to track impressions for, and we then insert references to them into an initial Add-Wins Set of advertisements.
- **Selection of displayable advertisements.** We define a dataflow computation in Lasp that will derive an Add-Wins Set of advertisements to display to the clients based on advertisements that have valid "contracts": records that represent that an advertisement is allowed to be displayed at the current time (Figure 1).
- **Enforcing invariants.** Since clients increment each advertisement counter as advertisement impressions occur, when the target number of impressions is reached both the client and the server will fire a trigger to remove the advertisement counter from the set of advertisements, to prevent the advertisement from being further displayed. This can be done without coordination through the use of the Add-Wins Set.

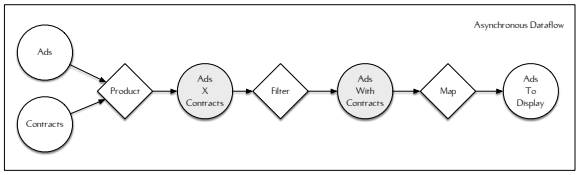


Figure 1: Asynchronous dataflow computation in Lasp that derives the set of displayable advertisements.

[2] Monotonicity implies that once a condition becomes true, it remains true; a monotonicity check can be done without distributed coordination.
[3] a.k.a. Observed-Remove Set

The advertisement counter has two important design choices, which makes its implementation in Lasp ideal.

- **Offline support.** As Angry Birds is a mobile application, there will be periods without connectivity. During this time, advertisements should still be displayable.
- **Lower-bound invariant.** Advertisements need to be displayed a minimum number of times; additional impressions are not problematic. This is a monotonic condition: once the condition is true, it remains true.

2.3 Implementation

The advertisement counter is broken into two components that work in concert. Both components track a single replica of a set of identifiers of displayable advertisements, and for each identifier a replica of an advertisement counter that tracks the total number of times the advertisement has been displayed to the user. Each node in our experiment runs either a single client or server process.

- **Server processes.** One or more server processes, each responsible for propagating their state to clients and disabling advertisements that have been displayed a minimum number of times by monotonically removing them from the set of displayable advertisements.
- **Client processes.** Many client processes that periodically propagate their state with other nodes, and increment their counter replicas based on a synthetic workload.

The prototype implementation of the Lasp programming model is built in the Erlang programming language and exposed to the user as an application library.

The fully instrumented Lasp advertisement counter client is implemented in 276 lines of Erlang code, and the fully instrumented advertisement counter server is 333 lines of Erlang code. Around 50% of this code is for instrumentation and orchestration, to ensure we can perform a full analysis of the application during experimentation. The Lasp runtime system takes care of cluster maintenance, data synchronization and storage, which are done manually in the previous approaches (ad-hoc merge or custom CRDT design).

3 SYSTEM ARCHITECTURE

To perform a real world evaluation of the advertisement counter, we implemented an efficient, scalable runtime system for Lasp. Lasp's runtime system is a highly-scalable eventually consistent data store with two different dissemination mechanisms (state-based vs. delta-based) and two different cluster topologies (datacenter vs. hybrid gossip). Lasp's programming model, presented in [11], sits above the data store and exposes a programming interface.

Datacenter Lasp [11] operates using a structured overlay network. Hybrid Gossip Lasp [12] uses an unstructured overlay network, and by design should achieve greater scalability and provide better fault-tolerance [15].

3.1 Datacenter Lasp

Datacenter Lasp refers to the prototype implementation of the runtime system presented with the programming model, at this conference two years ago [11].

In Datacenter Lasp, all CRDT state is both partitioned and replicated across several datacenter nodes. Client processes communicate directly with server processes that are running on datacenter nodes; client processes do not communicate amongst each other. Replication is used across datacenter nodes for fault tolerance, and partitioning/sharding is used for horizontal scalability: this is achieved through the use of consistent hashing and hash-space partitioning. In our experiments this is simplified and there is no partitioning, since the data set for our experiments never exceeds a single datacenter node's available capacity.

3.2 Hybrid Gossip Lasp

Hybrid Gossip Lasp is inspired by two Hybrid Gossip protocols, HyParView [10], and Plumtree [9]. In Hybrid Gossip Lasp, nodes are assembled in a peer-to-peer topology, where client processes can communicate either with server processes running on datacenter nodes or client processes. State is delivered transitively through other processes in the system: there is no need to communicate directly with a server process running on a datacenter node.

Hybrid Gossip Lasp uses a membership protocol heavily inspired by HyParView, to compute an overlay network containing all of the members in the cluster. The notable differences between the HyParView protocol and our membership protocol were the results of adapting the theoretical treatment in the HyParView paper to an actual implementation that was used for this experiment.

Specifically, the original HyParView protocol was evaluated in a low-churn environment, whereas our environment has much higher churn. *Churn* is defined as rate of node turnover, i.e., percentage of nodes leaving and being replaced by new nodes, per time unit. The higher churn in our environment was a byproduct of attempting to reduce experimentation time to save costs when operating large clusters: this allowed experiments that would normally take hours for cluster deployment and operations to be reduced to fractional hours at significant cost savings. For details on the modifications to the protocol, the reader is referred to [13].

3.3 Dissemination Protocols

The system supports two data dissemination protocols.

- **State-based.** Objects are locally updated through mutators that inflate the state. Objects are periodically sent to peers that merge the received object with their local state.
- **Delta-based.** Objects are locally updated by merging the state with the result of δ-mutators [1], called deltas, that compactly represent changed portions of state. These deltas are buffered locally and sent to each local peer in every propagation interval.

4 ENGINEERING SCALE

The Lasp semantics ensures that the runtime system is correct in theory for arbitrary distribution of the computation. However, engineering a scalable real-world system requires a significant amount of sophisticated tooling to ensure scalability both for deployment and for observability during execution. Near the end of the SyncFree project, we designed an experiment with the goal of scaling to 10 000 nodes. We finally achieved a scale of 1024 nodes at a total cloud computing cost of about €9000.

4.1 Experiment Configuration

For the purposes of the experiment, we used a total of 70 m3.2xlarge instances in the Amazon EC2 cloud computing environment, within the same region and availability zone. We used the Apache Mesos [7] cluster computing framework to subdivide each of these machines into smaller, fully-isolated machines using cgroups. Each virtual machine, representing a single Lasp node, communicated with other nodes in the cluster using TCP, and given the uniform deployment across all of the allocated instances, had varying latencies to other nodes in the system depending on their physical location.

When subdividing resources for the experiment, we allocated each server task 4 GB of memory with 2 virtual CPUs, and each client task 1 GB of memory, with 0.5 virtual CPUs. Here a *task* is a logical unit of computation that is executed on one virtual machine. We consider that these numbers vastly underrepresent the capabilities of modern mobile devices in widespread deployment today and therefore will lead to conservative results in the evaluation. We allocate more resources to servers, specifically in Datacenter Lasp mode, as servers are required to maintain connections to more nodes in the system; the advertisement counter does not require more resources between Datacenter and Hybrid Gossip modes.

4.2 Experimental Workflow

As running experiments in an unsimulated cloud environment can be challenging due to the inherent nondeterminism across different executions of the same experiment, we created a workflow targeted at reducing nondeterminism by controlling the experiments' setup and teardown procedures with detailed instrumentation for post-experimental analysis. We describe that workflow below.

- **Bootstrapping.** Initially, all of the server and client processes are bootstrapped and joined into a single cluster. The experiment does not begin until we ensure that all of the nodes in the system are connected and the connection graph forms a single connected component. Each node should be reachable by every other node in the system, either directly as a local neighbor, or indirectly via multi-hop. During this process, the system creates advertisement counters and the set of displayable ads.
- **Simulation.** Once we ensure the cluster is connected, each node starts collecting metrics and generating its own workload that randomly selects a counter to increment based on the set of displayable advertisements every predefined impression interval. Periodically, each process propagates local replicas with neighbor processes. It should be noted that each client has its own workload generator: using a centralized harness for running the experiment introduces coordination, which reduces the scalability of the system.
- **Convergence.** As each of the experiments has a controlled number of events that will be generated based on the number of clients participating in the system, the experiment continues to run until each node has observed the effects of all events: we refer to this process as convergence.
- **Metrics aggregation and archival.** Once convergence is reached, the experiment is complete. Each node, upon observing convergence begins uploading metrics recorded during the experiment to a central location: these logs are used for analysis of the runtime system. Once this process is complete, the experiment harness waits for the system to fully teardown the cluster before starting a subsequent run, to prevent state leakage between runs when reusing the same hardware to reduce costs.

4.3 Experimental Infrastructure

Evaluation of a large-scale distributed programming model is difficult. This is due to failures in the underlying frameworks that are used to provide mechanisms for deployment and operations, and because of inadequate tools required to observe the system during execution to ensure it is operating properly.

4.3.1 Apache Mesos. While experimentation shows Lasp scalability to 1024 nodes, we do not believe that this number is a firm upper limit. When attempting to run experiments with 2048 nodes we quickly ran into problems with the Apache Mesos cloud computing framework. One issue is that when attempting to bootstrap a cluster containing 70 instances too quickly, instances become disconnected and need to be manually reprovisioned. This required a slower cluster deployment where a cluster would be scaled from 35 instances, first to 50 instances, and then to 70 instances. As the 2048 experiment required 140 m3.2xlarge instances to operate, cluster deployment would take significantly longer.

When attempting to launch 2048 tasks in Mesos (with a single task representing a single application node), instances would become overloaded quickly and fail to respond to heartbeat messages: this triggered these instances being marked as offline by Mesos and the tasks orphaned. This would require restarting the experiment and reallocating the cluster to account for the lost tasks.

4.3.2 Sprinter. Once tasks were launched by Apache Mesos, we needed a mechanism for client processes to discover other client processes in the system and connect to them.

Therefore, we built an open source service discovery library called Sprinter that was used to fetch a list of running tasks from the Mesos framework, Marathon, and supply them to the system as targets to connect to. Sprinter also performs the following functions:

- **Graph analysis for connectedness.** Each node uploads its local membership view to Amazon S3. The first, lexicographically ordered, server periodically pulls this membership information and builds a local graph that is analyzed to determine if the graph contains all clients, and that the connection graph forms a single connected component.
- **Delay experiment for connectedness.** Based on graph analysis, the experiment's start is delayed until the connection graph forms a single connected component.
- **Periodic reconnection if isolated.** If a node becomes isolated from the cluster, it will rejoin the cluster, using the information provided by Marathon.

To assist in operator debugging of the experiments, a graphical tool was built to visualize the graph information from Sprinter along with extensive logging to the server node with information about cluster conditions.

4.3.3 Partisan. Distributed Erlang has known scalability problems when operated in the range of 50 or more nodes as it tracks full membership information in the cluster at each node and maintains

full connectivity between nodes using a single TCP connection that is used for both data transmission and heartbeat messages. Single connections are problematic because of head-of-line blocking when large messages are transmitted.

We knew that for the experiment to scale we would need: (1) to move away from Distributed Erlang, (2) to configure network topologies for both Datacenter Lasp and Hybrid Gossip Lasp in a single specification, and (3) to specify configurations at runtime without having to modify application code. To do this we built Partisan, an open source Erlang library that provides an alternative communication layer that eschews the use of Distributed Erlang. Partisan supports multiple network configurations and topologies: a client-server star topology, a full connectivity topology mirroring Distributed Erlang's, a static topology where per-node membership is explicitly maintained, and a random unstructured overlay membership protocol inspired by the HyParView membership protocol.

4.3.4 Workflow CRDT (W-CRDT). In our experiments, a central task could not be used to orchestrate the execution: early experiments demonstrated that the central task quickly became a bottleneck and slowed down execution to the speed of the central task. Therefore, we eliminated the central task.

However, without a central task performing orchestration, it becomes more difficult to control when nodes should perform certain actions. For example, after event generation is complete, we should wait for convergence before proceeding to metrics aggregation. Therefore, we needed a mechanism for asynchronously controlling the workflow of the application scenario.

We devised a novel data structure, called the *Workflow-CRDT* (W-CRDT), that is disseminated between nodes for controlling when certain actions should take place. This object is not instrumented by our runtime or included in any of the application logging, to prevent the structure itself from influencing the results of the experiment. The W-CRDT is a sequence of Grow-Only Map CRDTs, where each map is a function from opaque node identifiers to booleans. The sequence is implemented with the recursive Pair CRDT (similar to a recursive list type). The W-CRDT operates as follows:

- **Per node flag.** Each node's portion of a task to be completed is modeled as a flag; each node toggles its flag when it has completed its work.
- **Tasks as grow-only maps.** Each task that needs to be performed is represented by one grow-only map. When all the map's flags are true, the task is considered as complete. This corresponds to a barrier synchronization.
- **Sequential composition of tasks.** Each task can be sequenced with another task. A task starts when its preceding task has completed.
- **Workflow completion.** The workflow is considered complete when all of the tasks that make up the sequential composition are complete.

The W-CRDT is used to model the following sequential workflow in each experiment.

- **Perform event generation.** Once event generation is complete, nodes mark event generation complete.
- **Blocking for convergence.** Once convergence is reached, nodes mark convergence complete.
- **Log aggregation.** Once convergence is reached, nodes begin uploading their logs to a central location and mark log aggregation complete.
- **Shutdown.** Shutdown once log aggregation is complete.

5 EVALUATION

For Datacenter Lasp, we ran experiments using state-based dissemination, with a single server, and 32, 64, 128, 256 clients, forming with the server a star graph topology. For Hybrid Gossip Lasp, we ran experiments using both dissemination strategies, with a single server, and 32, 64, 128, 256, 512, and 1024 clients.

Each experiment was run twice, with the advertisement impression interval fixed at 10 seconds and the propagation interval at 5 seconds. The total number of impressions was configured to ensure that, in all executions, the experiment ran for 30 minutes.

Figure 2 and Figure 3 evaluate three different operational modes for Lasp, examining the state transmission for the duration of the experiment. Two Hybrid Gossip dissemination strategies, state-based and delta-based, are evaluated using a single overlay generated by the HyParView protocol. We also evaluated Datacenter Lasp, where clients propagate changes to the server using a state-based dissemination strategy. We did not evaluate delta-based for Datacenter Lasp, as it is unrealistic to believe that the server could buffer all changes in the system. In this evaluation, we scale up to 256 client processes: this is the largest number of client processes a single server could support in Datacenter Lasp. Hybrid Gossip scaled to 1024 nodes, before we ran into issues with Apache Mesos.

Datacenter Lasp performs the best in terms of state transmission when compared to Hybrid Gossip Lasp using the same dissemination strategy. This results from Datacenter Lasp have no redundancy at all: the star topology has a single point of failure that is used for communication between all nodes in the system. Delta-based dissemination demonstrates a clear advantage for Hybrid Gossip Lasp where redundancy is required to keep the system operating: state transmission can be reduced without sacrificing the fault-tolerance properties of the underlying overlay network. In terms of protocol transmission in Hybrid Gossip Lasp, delta-based dissemination performs better than state-based, even though it is a more complex protocol: in delta-based dissemination a process can track which updates have been seen by its neighbor processes and it will not disseminate an unchanged object, while in state-based dissemination an object is always propagated.

Our experiments confirm several design considerations made in Lasp. First, as demonstrated by the graphs, in the Datacenter Lasp model the transmission cost is reduced as there is no redundancy in messaging and subsequently no fault-tolerance. In this model, because of communication through a datacenter node, an update takes two hops to reach all clients in the system. However, this model has limited scalability because a centralized point, which could be partitioned and replicated across multiple servers, is used as a coordination point for all clients.

Hybrid Gossip Lasp adds additional redundancy by constructing a random overlay network using the HyParView protocol and gossiping state to local peers. This model has additional cost, but provides fault-tolerance through redundancy. In the worst case, an update will be observed by all nodes V after $\log |V|$ propagation

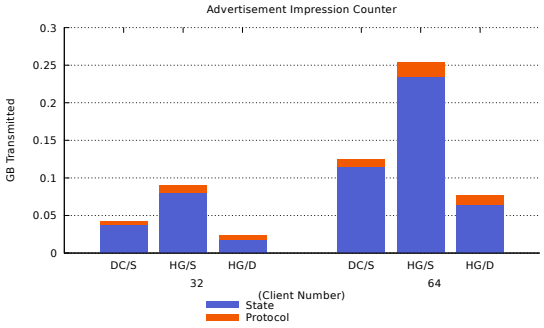


Figure 2: Comparison of state- and delta-based dissemination in both Datacenter and Hybrid Gossip Lasp with 32/64 clients.

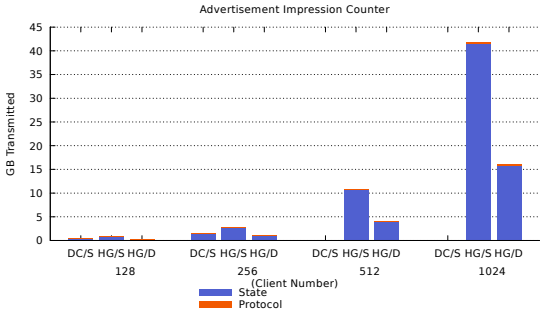


Figure 3: Comparison of state- and delta-based dissemination in both Datacenter and Hybrid Gossip Lasp with Datacenter Lasp ≤ 256 clients (limited in scalability) and Hybrid Gossip Lasp ≤ 1024 clients.

intervals, since in this topology the diameter is logarithmic on the number of nodes.

6 CONCLUSION

Designing new programming models for building large-scale distributed applications requires not only a solid theoretical design, but a well-engineered solution to demonstrate that the system can scale as advertised. Specifically, large-scale evaluations are plagued by the following problems.

- **Existing tooling can be problematic.** Existing infrastructure, frameworks, and languages can be treacherous as they can reduce the scalability of the system because of their design choices.
- **Visualizations are invaluable.** Visualizations assist in debugging the system in real time.
- **Achieving reproducibility is non-trivial.** Clouds provide high-level abstractions over machines, removing visibility into server location and isolation which makes controlled experiments difficult.
- **Performance can fluctuate.** Virtual machine placement and migration, compounded by a language VM layer, are factors that make performance measurement unpredictable. Cost considerations also limit the statistical smoothing possible by running multiple experiments.
- **Evaluations are expensive.** To provide a real world evaluation, significant funding is required for the infrastructure resources and significant time is required for developing deployment tools and for debugging experiments.

Lasp's scalable design was achieved by taking a holistic approach: both the runtime system and programming model were designed to accommodate one another in a way that allows scalability. However, the effort required to demonstrate Lasp as both scalable and practical remained a non-trivial challenge.

REFERENCES

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2016. Delta State Replicated Data Types. *CoRR* abs/1603.01529 (2016). http://arxiv.org/abs/1603.01529
[2] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. 249–260.
[3] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. 2015. Global sequence protocol: A robust abstraction for replicated shared state. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[4] Christopher S. Meiklejohn. 2017. Lasp Language Documentation. https://lasp-lang.org. (2017).
[5] Christopher S. Meiklejohn. 2017. Lasp Language Source Repository. https://github.com/lasp-lang. (2017).
[6] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 1.
[7] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.
[8] Lindsey Kuper and Ryan R Newton. 2014. Joining forces: toward a unified account of LVars and convergent replicated data types. In *5th Workshop on Determinism and Correctness in Parallel Programming (WoDet 2014)*.
[9] João Leitão, Jose Pereira, and Luìs Rodrigues. 2007. Epidemic broadcast trees. IEEE, 301–310.
[10] João Leitão, José Pereira, and Luìs Rodrigues. 2007. HyParView: A membership protocol for reliable gossip-based broadcast. IEEE, 419–429.
[11] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. ACM, 184–195.
[12] Christopher Meiklejohn and Peter Van Roy. 2015. Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation. In *Reliable Distributed Systems Workshop (SRDSW), 2015 IEEE 34th Symposium on*. IEEE, 62–67.
[13] Christopher S Meiklejohn and Peter Van Roy. 2017. Loquat: A framework for large-scale actor communication on edge networks. In *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*. IEEE, 563–568.
[14] Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. 2016. I now pronounce you reactive and consistent: handling distributed and replicated state in reactive programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems*. ACM, 1–8.
[15] Rodrigo Rodrigues and Peter Druschel. 2010. Peer-to-peer systems. *Commun. ACM* 53, 10 (2010), 72–82.
[16] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of convergent and commutative replicated data types*. Technical Report RR-7506. INRIA.
[17] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981
[18] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. *Managing update conflicts in Bayou, a weakly connected replicated storage system*. Vol. 29. ACM.

Non-uniform Replication

Gonçalo Cabrita[1] and Nuno Preguiça[2]

1 **NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal**
g.cabrita@campus.fct.unl.pt
2 **NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal**
nuno.preguica@fct.unl.pt

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 non-uniform 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.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Non-uniform Replication, Partial Replication, Replicated Data Types, Eventual Consistency

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.24

1 Introduction

Many applications run on cloud infrastructures composed by multiple data centers, geographically distributed across the world. These applications usually store their data on geo-replicated data stores, with replicas of data being maintained in multiple data centers. Data management in geo-replicated settings is challenging, requiring designers to make a number of choices to better address the requirements of applications.

One well-known trade-off is between availability and data consistency. Some data stores provide strong consistency [5, 17], where the system gives the illusion that a single replica exists. This requires replicas to coordinate for executing operations, with impact on the latency and availability of these systems. Other data stores [7, 11] provide high-availability and low latency by allowing operations to execute locally in a single data center eschewing a linearizable consistency model. These systems receive and execute updates in a single replica before asynchronously propagating the updates to other replicas, thus providing very low latency.

With the increase of the number of data centers available to applications and the amount of information maintained by applications, another trade-off is between the simplicity of maintaining all data in all data centers and the cost of doing so. Besides sharding data among multiple machines in each data center, it is often interesting to keep only part of the data in each data center to reduce the costs associated with data storage and running

protocols that involve a large number of replicas. In systems that adopt a partial replication model [22, 25, 6], as each replica only maintains part of the data, it can only locally process a subset of the database queries. Thus, when executing a query in a data center, it might be necessary to contact one or more remote data centers for computing the result of the query.

In this paper we explore 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 if a remove operation is available, as one of the elements not in the top needs to be promoted when a top element is removed.

A top-K object could be used for maintaining the leaderboard in an online game. In such system, while the information for each user only needs to be kept in the data center closest to the user (and in one or two more for fault tolerance), it is important to keep a replica of the leaderboard in every data center for low latency and availability. Currently, for supporting such a feature, several designs could be adopted. First, the system could maintain an object with the results of all players in all replicas. While simple, this approach turns out to be needlessly expensive in both storage space and network bandwidth when compared to our proposed model. Second, the system could move all data to a single data center and execute the computation in that data center or use a data processing system that can execute computations over geo-partitioned data [10]. The result would then have to be sent to all data centers. This approach is much more complex than our proposal, and while it might be interesting when complex machine learning computations are executed, it seems to be an overkill in a number of situations.

We apply the non-uniform replication model to eventual consistency and Conflict-free Replicated Data Types [23], formalizing the model for an operation-based replication approach. We present two useful data type designs that implement such model. Our evaluation shows that the non-uniform replication model leads to high gains in both storage space and network bandwidth used for synchronization when compared with state-of-the-art replication based alternatives.

In summary, this paper makes the following contributions:

- The proposal of the non-uniform replication model, where each replica only keeps part of the data but enough data to reply to every query;
- The definition of non-uniform eventual consistency (NuEC), the identification of sufficient conditions for providing NuEC and a protocol that enforces such conditions relying on operation-based synchronization;
- Two useful replicated data type designs that adopt the non-uniform replication model (and can be generalized to use different filter functions);
- An evaluation of the proposed model, showing its gains in term of storage space and network bandwidth.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 describes the non-uniform replication model. Section 4 applies the model to an eventual consistent system. Section 5 introduces two useful data type designs that follow the model. Section 6 compares our proposed data types against state-of-the-art CRDTs.

2 Related Work

Replication: A large number of replication protocols have been proposed in the last decades [8, 27, 15, 16, 2, 21, 17]. Regarding the contents of the replicas, these protocols can be divided in those providing full replication, where each replica maintains the full database state, and partial replication, where each replica maintains only a subset of the database state.

Full replication strategies allow operations to concurrently modify all replicas of a system and, assuming that replicas are mutually consistent, improves availability since clients may query any replica in the system and obtain an immediate response. While this improves the performance of read operations, update operations now negatively affect the performance of the system since they must modify every replica which severely affects middle-scale to large-scale systems in geo-distributed settings. This model also has the disadvantage of limiting the system's total capacity to the capacity of the node with fewest resources.

Partial replication [3, 22, 25, 6] addresses the shortcomings of full replication by having each replica store only part of the data (which continues being replicated in more than one node). This improves the scalability of the system but since each replica maintains only a part of the data, it can only locally process a subset of queries. This adds complexity to the query processing, with some queries requiring contacting multiple replicas to compute their result. In our work we address these limitations by proposing a model where each replica maintains only part of the data but can reply to any query.

Despite of adopting full or partial replication, replication protocols enforce strong consistency [17, 5, 18], weak consistency [27, 7, 15, 16, 2] or a mix of these consistency models [24, 14]. In this paper we show how to combine non-uniform replication with eventual consistency. An important aspect in systems that adopt eventual consistency is how the system handles concurrent operations. CRDTs have been proposed as a technique for addressing such challenge.

CRDTs: Conflict-free Replicated Data Types [23] are data types designed to be replicated at multiple replicas without requiring coordination for executing operations. CRDTs encode merge policies used to guarantee that all replicas converge to the same value after all updates are propagated to every replica. This allows an operation to execute immediately on any replica, with replicas synchronizing asynchronously. Thus, a system that uses CRDTs can provide low latency and high availability, despite faults and network latency. With these guarantees, CRDTs are a key building block for providing eventual consistency with well defined semantics, making it easier for programmers to reason about the system evolution.

When considering the synchronization process, two main types of CRDTs have been proposed: state-based CRDT, where replicas synchronize pairwise, by periodically exchanging the state of the replicas; and operation-based CRDTs, where all operations need to be propagated to all replicas.

Delta-based CRDTs [1] improve upon state-based CRDTs by reducing the dissemination cost of updates, sending only a delta of the modified state. This is achieved by using *delta-mutators*, which are functions that encode a delta of the state. Linde et. al [26] propose an improvement to delta-based CRDTs that further reduce the data that need to be propagated when a replica first synchronizes with some other replica. This is particularly interesting in peer-to-peer settings, where the synchronization partners of each replica change frequently. Although delta-based CRDTs reduce the network bandwidth used for synchronization, they continue to maintain a full replication strategy where the state of quiescent replicas is equivalent.

Computational CRDTs [19] are an extension of state-based CRDTs where the state of the object is the result of a computation (e.g. the average, the top-K elements) over the executed updates. As with the model we propose in this paper, replicas do not need to have equivalent states. The work we present in this paper extends the initial ideas proposed in computational CRDTs in several aspects, including the definition of the non-uniform replication model, its application to operation-based eventual consistency and the new data type designs.

3 Non-uniform replication

We consider an asynchronous distributed system composed by n nodes. Without loss of generality, we assume that the system replicates a single object. The object has an interface composed by a set of read-only operations, $\mathcal{Q}$, and a set of update operations, $\mathcal{U}$. Let $\mathcal{S}$ be the set of all possible object states, the state that results from executing operation o in state $s \in \mathcal{S}$ is denoted as $s \bullet o$. For a read-only operation, $q \in \mathcal{Q}$, $s \bullet q = s$. The result of operation $o \in \mathcal{Q} \cup \mathcal{U}$ in state $s \in \mathcal{S}$ is denoted as $o(s)$ (we assume that an update operation, besides modifying the state, can also return some result).

We denote the state of the replicated system as a tuple $(s_1, s_2, \ldots, s_n)$, with s_i the state of the replica i. The state of the replicas is synchronized by a replication protocol that exchanges messages among the nodes of the system and updates the state of the replicas. For now, we do not consider any specific replication protocol or strategy, as our proposal can be applied to different replication strategies.

We say a system is in a quiescent state for a given set of executed operations if the replication protocol has propagated all messages necessary to synchronize all replicas, i.e., additional messages sent by the replication protocol will not modify the state of the replicas. In general, replication protocols try to achieve a convergence property, in which the state of any two replicas is equivalent in a quiescent state.

▶ **Definition 1** (Equivalent state)**.** Two states, s_i and s_j, are *equivalent*, $s_i \equiv s_j$, iff the results of the execution of any sequence of operations in both states are equal, i.e., $\forall o_1, \ldots, o_n \in \mathcal{Q} \cup \mathcal{U}, o_n(s_i \bullet o_1 \bullet \ldots \bullet o_{n-1}) = o_n(s_j \bullet o_1 \bullet \ldots \bullet o_{n-1})$.

This property is enforced by most replication protocols, independently of whether they provide strong or weak consistency [13, 15, 27]. We note that this property does not require that the internal state of the replicas is the same, but only that the replicas always return the same results for any executed sequence of operations.

In this work, we propose to relax this property by requiring only that the execution of read-only operations return the same value. We name this property as *observable equivalence* and define it formally as follows.

▶ **Definition 2** (Observable equivalent state)**.** Two states, s_i and s_j, are *observable equivalent*, $s_i \overset{\circ}{\equiv} s_j$, iff the result of executing every read-only operation in both states is equal, i.e., $\forall o \in \mathcal{Q}, o(s_i) = o(s_j)$.

As read-only operations do not affect the state of a replica, the results of the execution of any sequence of read-only operations in two observable equivalent states will also be the same. We now define a non-uniform replication system as one that guarantees only that replicas converge to an observable equivalent state.

▶ **Definition 3** (Non-uniform replicated system)**.** We say that a replicated system is non-uniform if the replication protocol guarantees that in a quiescent state, the state of any two replicas is observable equivalent, i.e., in the quiescent state $(s_1, \ldots, s_n)$, we have $s_i \overset{\circ}{\equiv} s_j, \forall s_i, s_j \in \{s_1, \ldots, s_n\}$.

3.1 Example

We now give an example that shows the benefit of non-uniform replication. Consider an object *top-1* with three operations: (i) $add(name, value)$, an update operation that adds the pair to the top; (ii) $rmv(name)$, an update operation that removes all previously added pairs for *name*; (iii) $get()$, a query that returns the pair with the largest value (when more than one pair has the same largest value, the one with the smallest lexicographic name is returned).

Consider that $add(a, 100)$ is executed in a replica and replicated to all replicas. Later $add(b, 110)$ is executed and replicated. At this moment, all replicas know both pairs.

If later $add(c, 105)$ executes in some replica, the replication protocol does not need to propagate the update to the other replicas in a non-uniform replicated system. In this case, all replicas are observable equivalent, as a query executed at any replica returns the same correct value. This can have an important impact not only in the size of object replicas, as each replica will store only part of the data, but also in the bandwidth used by the replication protocol, as not all updates need to be propagated to all replicas.

We note that the states that result from the previous execution are not equivalent because after executing $rmv(b)$, the get operation will return $(c, 105)$ in the replica that has received the $add(c, 105)$ we operation and $(b, 100)$ in the other replicas.

Our definition only forces the states to be observable equivalent after the replication protocol becomes quiescent. Different protocols can be devised giving different guarantees. For example, for providing linearizability, the protocol should guarantee that all replicas return $(c, 105)$ after the remove. This can be achieved, for example, by replicating the now relevant $(c, 105)$ update in the process of executing the remove.

In the remainder of this paper, we study how to apply the concept of non-uniform replication in the context of eventually consistent systems. The study of its application to systems that provide strong consistency is left for future work.

4 Non-uniform eventual consistency

We now apply the concept of non-uniform replication to replicated systems providing eventual consistency.

4.1 System model

We consider an asynchronous distributed system composed by n nodes, where nodes may exhibit fail-stop faults but not byzantine faults. We assume a communication system with a single communication primitive, $mcast(m)$, that can be used by a process to send a message to every other process in the system with reliable broadcast semantics. A message sent by a correct process is eventually received by all correct processes. A message sent by a faulty process is either received by all correct processes or none. Several communication systems provide such properties – e.g. systems that propagate messages reliably using anti-entropy protocols [8, 9].

An object is defined as a tuple $(\mathcal{S}, s^0, \mathcal{Q}, \mathcal{U}_p, \mathcal{U}_e)$, where $\mathcal{S}$ is the set of valid states of the object, $s^0 \in \mathcal{S}$ is the initial state of the object, $\mathcal{Q}$ is the set of read-only operations (or *queries*), $\mathcal{U}_p$ is the set of prepare-update operations and $\mathcal{U}_e$ is the set of effect-update operations.

A query executes only at the replica where the operation is invoked, its source, and it has no side-effects, i.e., the state of an object remains unchanged after executing the operation.

When an application wants to update the state of the object, it issues a prepare-update operation, $u_p \in \mathcal{U}_p$. A u_p operation executes only at the source, has no side-effects and generates an effect-update operation, $u_e \in \mathcal{U}_e$. At source, u_e executes immediately after u_p.

As only effect-update operations may change the state of the object, for reasoning about the evolution of replicas we can restrict our analysis to these operations. To be precise, the execution of a prepare-update operation generates an instance of an effect-update operation. For simplicity, we refer the instances of operations simply as operations. With O_i the set of operations generated at node i, the set of operations generated in an execution, or simply the set of operations in an execution, is $O = O_1 \cup \ldots \cup O_n$.

4.2 Non-uniform eventual consistency

For any given execution, with O the operations of the execution, we say a replicated system provides *eventual consistency* iff in a quiescent state: (i) every replica executed all operations of O; and (ii) the state of any pair of replicas is equivalent.

A sufficient condition for achieving the first property is to propagate all generated operations using reliable broadcast (and execute any received operation). A sufficient condition for achieving the second property is to have only commutative operations. Thus, if all operations commute with each other, the execution of any serialization of O in the initial state of the object leads to an equivalent state.

From now on, unless stated otherwise, we assume that all operations commute. In this case, as all serializations of O are equivalent, we denote the execution of a serialization of O in state s simply as $s \bullet O$.

For any given execution, with O the operations of the execution, we say 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. 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 $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet O_{core}$. We define the set of operations that are irrelevant to the final state of the replicas as follows: $O_{masked} \subseteq O$ is a set of masked operations of O iff $s^0 \bullet O \stackrel{\circ}{\equiv} s^0 \bullet (O \setminus O_{masked})$.

▶ **Theorem 4** (Sufficient conditions for NuEC). *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.*

Proof. From the definition of core operations of O, and by the fact that all operations commute, it follows immediately that if a replica executes a set of core operations, then the final state of the replica is observable equivalent to the state obtained by executing a serialization of O. Additionally, any replica reaches an observable equivalent state. ◀

4.3 Protocol for non-uniform eventual consistency

We now build on the sufficient conditions for providing *non-uniform eventual consistency* to devise a correct replication protocol that tries to minimize the operations propagated to other replicas. The key idea is to avoid propagating operations that are part of a masked set. The challenge is to achieve this by using only local information, which includes only a subset of the executed operations.

Algorithm 1 presents the pseudo-code of an algorithm for achieving *non-uniform eventual consistency* – the algorithm does not address the durability of operations, which will be discussed later.

Algorithm 1 Replication algorithm for non-uniform eventual consistency

```
1:  S : state: initial s^0                                          ▷ Object state
2:  log_recv : set of operations: initial {}
3:  log_local : set of operations: initial {}                       ▷ Local operations not propagated
4:
5:  execOp(op): void                                                ▷ New operation generated locally
6:    log_local = log_local ∪ {op}
7:    S = S • op
8:
9:  opsToPropagate(): set of operations ▷ Computes the local operations that need to be propagated
10:   ops = maskedForever(log_local, S, log_recv)
11:   log_local = log_local \ ops
12:   opsImpact = hasObservableImpact(log_local, S, log_recv)
13:   opsPotImpact = mayHaveObservableImpact(log_local, S, log_recv)
14:   return opsImpact ∪ opsPotImpact
15:
16: sync(): void                                                    ▷ Propagates local operations to remote replicas
17:   ops = opsToPropagate()
18:   compactedOps = compact(ops)                                   ▷ Compacts the set of operations
19:   mcast(compactedOps)
20:   log_coreLocal = {}
21:   log_local = log_local \ ops
22:   log_recv = log_recv ∪ ops
23:
24: on receive(ops): void                                           ▷ Process remote operations
25:   log_recv = log_recv ∪ ops
26:   S = S • ops
```

The algorithm maintains the state of the object and two sets of operations: log_{local}, the set of effect-update operations generated in the local replica and not yet propagated to other replicas; log_{recv}, the set of effect-update operations propagated to all replicas (including operations generated locally and remotely).

When an effect-update operation is generated, the *execOp* function is called. This function adds the new operation to the log of local operations and updates the local object state.

The function *sync* is called to propagate local operations to remote replicas. It starts by computing which new operations need to be propagated, compacts the resulting set of operations for efficiency purposes, multicasts the compacted set of operations, and finally updates the local sets of operations. When a replica receives a set of operations (line 24), the set of operations propagated to all nodes and the local object state are updated accordingly.

Function *opsToPropagate* addresses the key challenge of deciding which operations need to be propagated to other replicas. To this end, we divide the operations in four groups.

First, the *forever masked* operations, which are operations that will remain in the set of masked operations independently of the operations that might be executed in the future. In the top example, an operation that adds a pair masks forever all known operations that added a pair for the same element with a lower value. These operations are removed from the set of local operations.

Second, the *core* operations (*opsImpact*, line 12), as computed locally. These operations need to be propagated, as they will (typically) impact the observable state at every replica.

Third, the operations that might impact the observable state when considered in combination with other non-core operations that might have been executed in other replicas (*opsPotImpact*, line 13). As there is no way to know which non-core operations have been executed in other replicas, it is necessary to propagate these operations also. For example, consider a modified top object where the value associated with each element is the sum of the values of the pairs added to the object. In this case, an add operation that would not move an element to the top in a replica would be in this category because it could influence

the top when combined with other concurrent adds for the same element.

Fourth, the remaining operations that might impact the observable state in the future, depending on the evolution of the observable state. These operations remain in log_{local}. In the original top example, an operation that adds a pair that will not be in the top, as computed locally, is in this category as it might become the top element after removing the elements with larger values.

For proving that the algorithm can be used to provide non-uniform eventual consistency, we need to prove the following property.

▶ **Theorem 5.** *Algorithm 1 guarantees that in a quiescent state, considering all operations O in an execution, all replicas have received all operations in a core set O_{core}.*

Proof. To prove this property, we need to prove that there exists no operation that has not been propagated by some replica and that is required for any O_{core} set. Operations in the first category have been identified as masked operations independently of any other operations that might have been or will be executed. Thus, by definition of masked operations, a O_{core} set will not (need to) include these operations. The fourth category includes operations that do not influence the observable state when considering all executed operations – if they might have impact, they would be in the third category. Thus, these operations do not need to be in a O_{core} set. All other operations are propagated to all replicas. Thus, in a quiescent state, every replica has received all operations that impact the observable state. ◀

4.4 Fault-tolerance

Non-uniform replication aims at reducing the cost of communication and the size of replicas, by avoiding propagating operations that do not influence the observable state of the object. This raises the question of the durability of operations that are not immediately propagated to all replicas.

One way to solve this problem is to have the source replica propagating every local operation to f more replicas to tolerate f faults. This ensures that an operation survives even in the case of f faults. We note that it would be necessary to adapt the proposed algorithm, so that in the case a replica receives an operation for durability reasons, it would propagate the operation to other replicas if the source replica fails. This can be achieved by considering it as any local operation (and introducing a mechanism to filter duplicate reception of operations).

4.5 Causal consistency

Causal consistency is a popular consistency model for replicated systems [15, 2, 16], in which a replica only executes an operation after executing all operations that causally precede it [12]. In the non-uniform replication model, it is impossible to strictly adhere to this definition because some operations are not propagated (immediately), which would prevent all later operations from executing.

An alternative would be to restrict the dependencies to the execution of core operations. The problem with this is that the status of an operation may change by the execution of another operation. When a non-core operation becomes core, a number of dependencies that should have been enforced might have been missed in some replicas.

We argue that the main interest of causal consistency, when compared with eventual consistency, lies in the semantics provided by the object. Thus, in the designs that we present in the next section, we aim to guarantee that in a quiescent state, the state of the replicated objects provide equivalent semantics to that of a system that enforces causal consistency.

5 Non-uniform operation-based CRDTs

CRDTs [23] are data-types that can be replicated, modified concurrently without coordination and guarantee the eventual consistency of replicas given that all updates propagate to all replicas. We now present the design of two useful operation-based CRDTs [23] that adopt the non-uniform replication model. Unlike most operation-based CRDT designs, we do not assume that the system propagates operations in a causal order. These designs were inspired by the state-based computational CRDTs proposed by Navalho *et al.* [19], which also allow replicas to diverge in their quiescent state.

5.1 Top-K with removals NuCRDT

In this section we present the design of a non-uniform top-K CRDT, as the one introduced in section 3.1. The data type allows access to the top-K elements added and can be used, for example, for maintaining the leaderboard in online games. The proposed design could be adapted to define any CRDT that filters elements based on a deterministic function by replacing the *topK* function used in the algorithm by another filter function.

For defining the semantics of our data type, we start by defining the happens-before relation among operations. To this end, we start by considering the happens-before relation established among the events in the execution of the replicated system [12]. The events that are considered relevant are: the generation of an operation at the source replica, and the dispatch and reception of a message with a new operation or information that no new message exists. We say that operation op_i happens before operation op_j iff the generation of op_i happened before the generation of op_j in the partial order of events.

The semantics of the operations defined in the top-K CRDT is the following. The *add(el,val)* operation adds a new pair to the object. The *rmv(el)* operation removes any pair of *el* that was added by an operation that happened-before the *rmv* (note that this includes non-core add operations that have not been propagated to the source replica of the remove). This leads to an *add-wins* policy [23], where a remove has no impact on concurrent adds. The *get()* operation returns the top-K pairs in the object, as defined by the function *topK* used in the algorithm.

Algorithm 2 presents a design that implements this semantics. The prepare-update *add* operation generates an effect-update *add* that has an additional parameter consisting in a timestamp $(replicaid, val)$, with *val* a monotonically increasing integer. The prepare-update *rmv* operation generates an effect-update *rmv* that includes an additional parameter consisting in a vector clock that summarizes add operations that happened before the remove operation. To this end, the object maintains a vector clock that is updated when a new add is generated or executed locally. Additionally, this vector clock should be updated whenever a replica receives a message from a remote replica (to summarize also the adds known in the sender that have not been propagated to this replica).

Besides this vector clock, *vc*, each object replica maintains: (i) a set, *elems*, with the elements added by all *add* operations known locally (and that have not been removed yet); and (ii) a map, *removes*, that maps each element *id* to a vector clock with a summary of the add operations that happened before all removes of *id* (for simplifying the presentation of the algorithm, we assume that a key absent from the map has associated a default vector clock consisting of zeros for every replica).

The execution of an *add* consists in adding the element to the set of *elems* if the add has not happened before a previously received remove for the same element – this can happen as operations are not necessarily propagated in causal order. The execution of a *rmv* consists

Algorithm 2 Top-K NuCRDT with removals

```
1:  elems : set of ⟨id, score, ts⟩ : initial {}
2:  removes : map id ↦ vectorClock: initial []
3:  vc : vectorClock: initial []
4:
5:  GET() : set
6:    return {⟨id, score⟩ : ⟨id, score, ts⟩ ∈ topK(elems)}
7:
8:  prepare ADD(id, score)
9:    generate add(id, score, ⟨getReplicaId(), ++ vc[getReplicaId()]⟩)
10:
11: effect ADD(id, score, ts)
12:   if removes[id][ts.siteId] < ts.val then
13:     elems = elems ∪ {⟨id, score, ts⟩}
14:     vc[ts.siteId] = max(vc[ts.siteId], ts.val)
15:
16: prepare RMV(id)
17:   generate rmv(id,vc)
18:
19: effect RMV(id, vc_rmv)
20:   removes[id] = pointwiseMax(removes[id], vc_rmv)
21:   elems = elems \ {⟨id_0, score, ts⟩ ∈ elem : id = id_0 ∧ ts.val ≤ vc_rmv[ts.siteId]}
22:
23: maskedForever(log_local, S, log_recv): set of operations
24:   adds = {add(id_1, score_1, ts_1) ∈ log_local :
25:     (∃add(id_2, score_2, ts_2) ∈ log_local : id_1 = id_2 ∧ score_1 < score_2 ∧ ts_1.val < ts_2.val)∨
26:       (∃rmv(id_3, vc_rmv) ∈ (log_recv ∪ log_local) : id_1 = id_3 ∧ ts_1.val ≤ vc_rmv[ts_1.siteId]}
27:   rmvs = {rmv(id_1, vc_1) ∈ log_local : ∃rmv(id_2, vc_2) ∈ (log_local ∪ log_recv) : id_1 = id_2 ∧ vc_1 < vc_2}
28:   return adds ∪ rmvs
29:
30: mayHaveObservableImpact(log_local, S, log_recv): set of operations
31:   return {}                          ▷ This case never happens for this data type
32:
33: hasObservableImpact(log_local, S, log_recv): set of operations
34:   adds = {add(id_1, score_1, ts_1) ∈ log_local : ⟨id_1, score_1, ts_1⟩ ∈ topK(S.elems)}
35:   rmvs = {rmv(id_1, vc_1) ∈ log_local : (∃add(id_2, score_2, ts_2) ∈ (log_local ∪ log_recv) :
36:       ⟨id_2, score_2, ts_2⟩ ∈ topK(S.elems ∪ {⟨id_2, score_2, ts_2⟩}) ∧ id_1 = id_2 ∧ ts_2.val ≤ vc_1[ts_2.siteId])}
37:   return adds ∪ rmvs
38:
39: compact(ops): set of operations
40:   return ops                         ▷ This data type does not require compaction
```

in updating *removes* and deleting from *elems* the information for adds of the element that happened before the remove. To verify if an add has happened before a remove, we check if the timestamp associated with the add is reflected in the remove vector clock of the element (lines 12 and 21). This ensures the intended semantics for the CRDT, assuming that the functions used by the protocol are correct.

We now analyze the code of these functions.

Function maskedForever computes: the local adds that become masked by other local adds (those for the same element with a lower value) and removes (those for the same element that happened before the remove); the local removes that become masked by other removes (those for the same element that have a smaller vector clock). In the latter case, it is immediate that a remove with a smaller vector clock becomes irrelevant after executing the one with a larger vector clock. In the former case, a local add for an element is masked by a more recent local add for the same element but with a larger value as it is not possible to remove only the effects of the later add without removing the effect of the older one. A local add also becomes permanently masked by a local or remote remove that happened after the add.

Function mayHaveObservableImpact returns the empty set, as for having impact on

any observable state, an operation also has to have impact on the local observable state by itself.

Function hasObservableImpact computes the local operations that are relevant for computing the top-K. An add is relevant if the added value is in the top; a remove is relevant if it removes an add that would be otherwise in the top.

5.2 Top Sum NuCRDT

We now present the design of a non-uniform CRDT, Top Sum, 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).

The semantics of the operations defined in the Top Sum object is the following. The *add(id, n)* update operation increments the value associated with *id* by *n*. The *get()* read-only operation returns the top-K mappings, $id \rightarrow value$, as defined by the *topK* function (similar to the Top-K NuCRDT).

This design is challenging, as it is hard to know which operations may have impact in the observable state. For example, consider a scenario with two replicas, where the value of the last element in the top is 100. If the known score of an element is 90, an add of 5 received in one replica may have impact in the observable state if the other replica has also received an add of 5 or more. One approach would be to propagate these operations, but this would lead to propagating all operations.

To try to minimize the number of operations propagated we use the following heuristic inspired by the demarcation protocol and escrow transactions [4, 20]. For each id that does not belong to the top, we compute the difference between the smallest value in the top and the value of the id computed by operations known in every replica – this is how much must be added to the id to make it to the top: let d be this value. If the sum of local adds for the id does not exceed $\frac{d}{num.replicas}$ in any replica, the value of id when considering adds executed in all replicas is smaller than the smallest element in the top. Thus, it is not necessary to propagate add operations in this case, as they will not affect the top.

Algorithm 3 presents a design that implements this approach. The state of the object is a single variable, *state*, that maps identifiers to their current values. The only prepare-update operation, *add*, generates an effect-update *add* with the same parameters. The execution of an effect-update *add(id, n)* simply increments the value of *id* by *n*.

Function maskedForever returns the empty set, as operations in this design can never be forever masked.

Function mayHaveObservableImpact computes the set of *add* operations that can potentially have an impact on the observable state, using the approach previously explained.

Function hasObservableImpact computes the set of *add* operations that have their corresponding *id* present in the top-K. This guarantees that the values of the elements in the top are kept up-to-date, reflecting all executed operations.

Function compact takes a set of *add* operations and compacts the *add* operations that affect the same identifier into a single operation. This reduces the size of the messages sent through the network and is similar to the optimization obtained in delta-based CRDTs [1].

Algorithm 3 Top Sum NuCRDT

```
1: state : map id ↦ sum: initial []
2:
3: GET() : map
4:   return topK(state)
5:
6: prepare ADD(id, n)
7:   generate add(id, n)
8:
9: effect ADD(id, n)
10:  state[id] = state[id] + n
11:
12: MASKEDFOREVER(log_local, S, log_recv): set of operations
13:  return {}                          ▷ This case never happens for this data type
14:
15: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
16:  top = topK(S.state)
17:  adds = {add(id, _) ∈ log_local : s = sum_val({add(i, n) ∈ log_local : i = id})
18:    ∧ s ≥ ((min(sum(top)) − (S.state[id] − s)) / getNumReplicas())}
19:  return adds
20:
21: HASOBSERVABLEIMPACT(log_local, S, log_recv): set of operations
22:  top = topK(S.state)
23:  adds = {add(id, _) ∈ log_local : id ∈ ids(top)}
24:  return adds
25:
26: COMPACT(ops): set of operations
27:  adds = {add(id, n) : id ∈ {i : add(i, _) ∈ ops}∧ n = sum({k : add(id_1, k) ∈ ops : id_1 = id})}
28:  return adds
```

5.3 Discussion

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. The Top-K CRDT presented in section 5.1 is an example of the former. 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. The Top-Sum CRDT presented in section 5.2 is an example of the latter. 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 [19].

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.

6 Evaluation

In this section we evaluate our data types that follow the non-uniform replication model. To this end, we compare our designs against state-of-the-art CRDT alternatives: delta-based CRDTs [1] that maintain full object replicas efficiently by propagating updates as deltas of the state; and computational CRDTs [19] that maintain non-uniform replicas using a state-based approach.

Our evaluation is performed by simulation, using a discrete event simulator. To show the benefit in terms of bandwidth and storage, we measure the total size of messages sent between replicas for synchronization (total payload) and the average size of replicas.

We simulate a system with 5 replicas for each object. Both our designs and the computational CRDTs support up to 2 replica faults by propagating all operations to, at least, 2 other replicas besides the source replica. We note that this limits the improvement that our approach could achieve, as it is only possible to avoid sending an operation to two of the five replicas. By either increasing the number of replicas or reducing the fault tolerance level, we could expect that our approach would perform comparatively better than the delta-based CRDTs.

6.1 Top-K with removals

We begin by comparing our Top-K design (*NuCRDT*) with a delta-based CRDT set [1] (*Delta CRDT*) and the top-K state-based computational CRDT design [19] (*CCRDT*).

The top-K was configured with K equal to 100. In each run, 500000 update operations were generated for 10000 Ids and with scores up to 250000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

Given the expected usage of top-K for supporting a leaderboard, we expect the remove to be an infrequent operation (to be used only when a user is removed from the game). Figures 1 and 2 show the results for workloads with 5% and 0.05% of removes respectively (the other operations are adds).

In both workloads our design achieves a significantly lower bandwidth cost when compared to the alternatives. The reason for this is that our design only propagates operations that will be part of the top-K. In the delta-based CRDT, each replica propagates all new updates and not only those that are part of the top. In the computational CRDT design, every time the top is modified, the new top is propagated. Additionally, the proposed design of computational CRDTs always propagates removes.

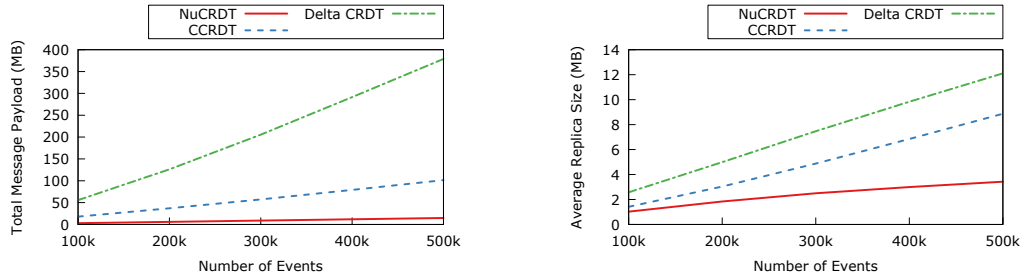


Figure 1 Top-K with removals: payload size and replica size, workload of 95/5

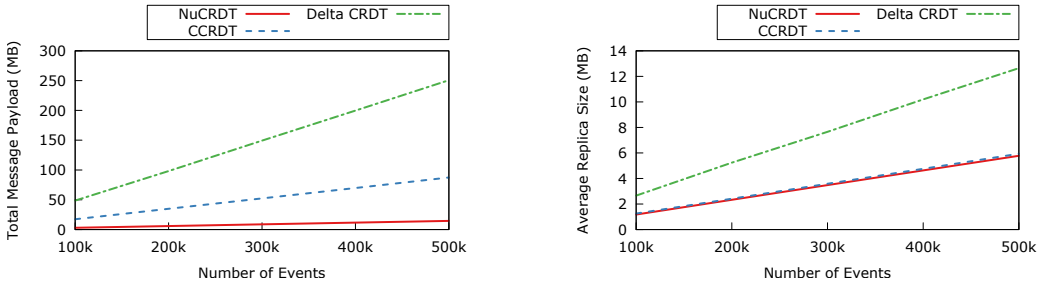


Figure 2 Top-K with removals: payload size and replica size, workload of 99.95/0.05

The results for the replica size show that our design is also more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information from remote operations received for guaranteeing fault-tolerance and those that have influenced the top-K at some moment in the execution. The computational CRDT design additionally keeps information about all removes. The delta-based CRDT keeps information about all elements that have not been removed or overwritten by a larger value. We note that as the percentage of removes approaches zero, the replica sizes of our design and that of computational CRDT starts to converge to the same value. The reason for this is that the information maintained in both designs is similar and our more efficient handling of removes starts becoming irrelevant. The opposite is also true: as the number of removes increases, our design becomes even more space efficient when compared to the computational CRDT.

6.2 Top Sum

To evaluate our Top Sum design (*NuCRDT*), we compare it against a delta-based CRDT map (*Delta CRDT*) and a state-based computational CRDT implementing the same semantics (*CCRDT*).

The top is configured to display a maximum of 100 entries. In each run, 500000 update operations were generated for 10000 Ids and with challenges awarding scores up to 1000. The values used in each operation were randomly selected using a uniform distribution. A replica synchronizes after executing 100 events.

Figure 3 shows the results of our evaluation. Our design achieves a significantly lower bandwidth cost when compared with the computational CRDT, because in the computational CRDT design, every time the top is modified, the new top is propagated. When compared with the delta-based CRDTs, the bandwidth of NuCRDT is approximately 55% of the bandwidth used by delta-based CRDTs. As delta-based CRDTs also include a mechanism for compacting propagated updates, the improvement comes from the mechanisms for avoiding

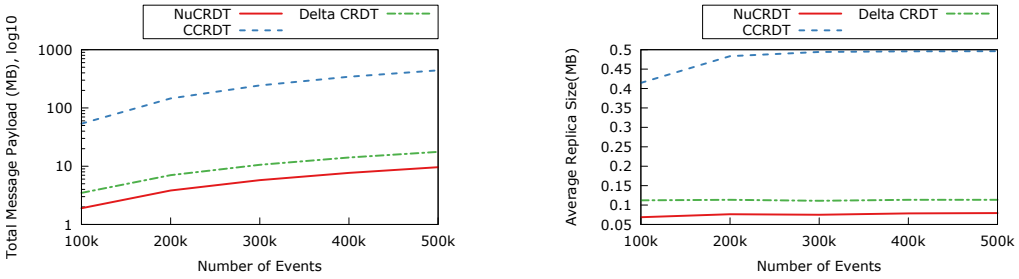


Figure 3 Top Sum: payload size and replica size

propagating operations that will not affect the top elements, resulting in less messages being sent.

The results for the replica size show that our design also manages to be more space efficient than previous designs. This is a consequence of the fact that each replica, besides maintaining information about local operations, only keeps information of remote operations received for guaranteeing fault-tolerance and those that have influenced the top elements at some moment in the execution.

7 Conclusions

In this paper we proposed the non-uniform replication model, an alternative model for replication that combines the advantages of both full replication, by allowing any replica to reply to a query, and partial replication, by requiring that each replica keeps only part of the data. We have shown how to apply this model to eventual consistency, and proposed a generic operation-based synchronization protocol for providing non-uniform replication. We further presented the designs of two useful replicated data types, the Top-K and Top Sum, that adopt this model (in appendix, we present two additional designs: Top-K without removals and Histogram). Our evaluation shows that the application of this new replication model helps to reduce the message dissemination costs and the size of replicas.

In the future we plan to study which other data types can be designed that adopt this model and to study how to integrate these data types in cloud-based databases. We also want to study how the model can be applied to strongly consistent systems.

Acknowledgments

This work has been partially funded by CMU-Portugal research project GoLocal Ref. CMUP-ERI/TIC/0046/2014, EU LightKone (grant agreement n.732505) and by FCT/MCT project NOVA-LINCS Ref. UID/CEC/04516/2013. Part of the computing resources used in this research were provided by a Microsoft Azure Research Award.

References

1 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *J. Parallel Distrib. Comput.*, 111:162–173, 2018. doi:10.1016/j.jpdc.2017.08.003.

2 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. doi:10.1145/2465351.2465361.

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

4 Daniel Barbará-Millá and Hector Garcia-Molina. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal*, 3(3):325–353, July 1994. doi:10.1007/BF01232643.

5 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.

6 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. doi:10.1145/2745947.2745953.

7 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. doi:10.1145/1294261.1294281.

8 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. doi:10.1145/41840.41841.

9 Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulieacute;. Epidemic Information Dissemination in Distributed Systems. *Computer*, 37(5), May 2004. doi:10.1109/MC.2004.1297243.

10 Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. Pixida: Optimizing Data Parallel Jobs in Wide-area Data Analytics. *Proc. VLDB Endow.*, 9(2):72–83, October 2015. doi:10.14778/2850578.2850582.

11 Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010. doi:10.1145/1773912.1773922.

12 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), July 1978. doi:10.1145/359545.359563.

13 Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2), May 1998. doi:10.1145/279227.279229.

14 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.

15 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proc. 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011. doi:10.1145/2043556.2043593.

16 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proc. 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

17 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. doi:10.14778/2536360.2536366.

18 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. doi:10.1145/3038912.3052603.

19 David Navalho, Sérgio Duarte, and Nuno Preguiça. A Study of CRDTs That Do Computations. In *Proc. 1st Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, 2015. doi:10.1145/2745947.2745948.

20 Patrick E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986. URL: http://doi.acm.org/10.1145/7239.7265, doi:10.1145/7239.7265.

21 Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1), March 2005. doi:10.1145/1057977.1057980.

22 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-Store: Genuine Partial Replication in Wide Area Networks. In *Proc. 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, 2010. doi:10.1109/SRDS.2010.32.

23 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proc. 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, 2011.

24 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. doi:10.1145/2043556.2043592.

25 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. doi:10.1145/2517349.2522731.

26 Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-crdts: Making δ-crdts delta-based. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, pages 12:1–12:4, 2016. doi:10.1145/2911151.2911163.

27 Werner Vogels. Eventually Consistent. *Commun. ACM*, 52(1), January 2009. doi:10.1145/1435417.1435432.

A APPENDIX

In this appendix we present two additional NuCRDT designs. These designs exemplify the use of different techniques for the creation of NuCRDTs.

A.1 Top-K without removals

A simpler example of a data type that fits our proposed replication model is a plain top-K, without support for the remove operation. This data type allows access to the top-K elements added to the object and can be used, for example, for maintaining a leaderboard in an online game. The top-K defines only one update operation, *add(id,score)*, which adds element *id* with score *score*. The *get()* operation simply returns the K elements with largest scores. Since the data type does not support removals, and elements added to the top-K which do not fit will simply be discarded this means the only case where operations have an impact in the observable state are if they are core operations – i.e. they are part of the top-K. This greatly simplifies the non-uniform replication model for the data type.

Algorithm 4 Top-K NuCRDT

```
1:  elems : {⟨id, score⟩} : initial {}
2:
3:  GET() : set
4:    return elems
5:
6:  prepare ADD(id, score)
7:    generate add(id, score)
8:
9:  effect ADD(id, score)
10:   elems = topK(elems ∪ {⟨id, score⟩})
11:
12: MASKEDFOREVER(log_local, S, log_recv) : set of operations
13:   adds = {add(id_1, score_1) ∈ log_local : (∃add(id_2, score_2) ∈ log_recv : id_1 = id_2 ∧ score_2 > score_1)
14:   return adds
15:
16: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
17:   return {}                                   ▷ Not required for this data type
18:
19: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
20:   return {add(id, score) ∈ log_local : ⟨id, score⟩ ∈ S.elems}
21:
22: COMPACT(ops): set of operations
23:   return ops                    ▷ This data type does not use compaction
```

Algorithm 4 presents the design of the top-K NuCRDT. The prepare-update *add(id,score)* generates an effect-update *add(id,score)*.

Each object replica maintains only a set of K tuples, *elems*, with each tuple being composed of an *id* and a *score*. The execution of *add(id,score)* inserts the element into the set, *elems*, and computes the top-K of *elems* using the function *topK*. The order used for the *topK* computation is as follows: $\langle id_1, score_1\rangle > \langle id_2, score_2\rangle$ iff $score_1 > score_2 \vee (score_1 = score_2 \wedge id_1 > id_2)$. We note that the *topK* function returns only one tuple for each element *id*.

Function MASKEDFOREVER computes the adds that become masked by other add operations for the same *id* that are larger according to the defined ordering. Due to the way the top is computed, the lower values for some given *id* will never be part of the top. Function MAYHAVEOBSERVABLEIMPACT always returns the empty set since operations in this data type are always core or forever masked. Function HASOBSERVABLEIMPACT returns the set of unpropagated add operations which add elements that are part of the top – essentially, the

add operations that are core at the time of propagation. Function COMPACT simply returns the given *ops* since the design does not require compaction.

A.2 Histogram

We now introduce the Histogram NuCRDT that maintains a histogram of values added to the object. To this end, the data type maintains a mapping of bins to integers and can be used to maintain a voting system on a website. The semantics of the operations defined in the histogram is the following: *add(n)* increments the bin n by 1; *merge(histogram$_{delta}$)* adds the information of a histogram into the local histogram; *get()* returns the current histogram.

Algorithm 5 Histogram NuCRDT

```
 1: histogram : map bin ↦ n : initial []
 2:
 3: GET() : map
 4:   return histogram
 5:
 6: prepare ADD(bin)
 7:   generate merge([bin ↦ 1])
 8:
 9: prepare MERGE(histogram)
10:   generate merge(histogram)
11:
12: effect MERGE(histogram_delta)
13:   histogram = pointwiseSum(histogram, histogram_delta)
14:
15: MASKEDFOREVER(log_local, S, log_recv) : set of operations
16:   return {}                                  ▷ Not required for this data type
17:
18: MAYHAVEOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
19:   return {}                                  ▷ Not required for this data type
20:
21: HASOBSERVABLEIMPACT(log_local, S, log_recv) : set of operations
22:   return log_local
23:
24: COMPACT(ops): set of operations
25:   deltas = {hist : merge(hist_delta) ∈ ops}
26:   return {merge(pointwiseSum(deltas))}
```

This data type is implemented in the design presented in Algorithm 5. The prepare-update *add(n)* generates an effect-update *merge([$n \mapsto 1$])*. The prepare-update operation *merge(histogram)* generates an effect-update *merge(histogram)*.

Each object replica maintains only a map, *histogram*, which maps *bins* to integers. The execution of a *merge(histogram$_{delta}$)* consists of doing a pointwise sum of the local histogram with *histogram$_{delta}$*.

Functions MASKEDFOREVER and MAYHAVEOBSERVABLEIMPACT always return the empty set since operations in this data type are always core. Function HASOBSERVABLEIMPACT simply returns log_{local}, as all operations are core in this data type. Function COMPACT takes a set of instances of *merge* operations and joins the histograms together returning a set containing only one *merge* operation.

Fine-Grained Consistency Upgrades for Online Services

Filipe Freitas[‡†⋆], João Leitão[†], Nuno Preguiça[†], and Rodrigo Rodrigues[⋆⋆]
[‡]ISEL, Instituto Superior de Engenharia de Lisboa, Portugal; [†]NOVA-LINCS & FCT,
Universidade NOVA de Lisboa, Portugal; [⋆]INESC-ID; [⋆]IST, Universidade de Lisboa, Portugal

***Abstract*—Online services such as Facebook or Twitter have public APIs to enable an easy integration of these services with third party applications. However, the developers who design these applications have no information about the consistency provided by these services, which exacerbates the complexity of reasoning about the semantics of the applications they are developing. In this paper, we show that is possible to deploy a transparent middleware between the application and the service, which enables a fine-grained control over the session guarantees that comprise the consistency semantics provided by these APIs, without having to gain access to the implementation of the underlying services. We evaluated our middleware using the Facebook public API and the Redis datastore, and our results show that we are able to provide fine-grained control of the consistency semantics incurring in a small local storage and modest latency overhead.**

I. Introduction

Many computer systems and applications make use of stateful services that run in the cloud, with various types of interfaces mediating the access to these cloud services. For instance, an application may decide to store its persistent state in a Cassandra cluster running on Azure instances, or directly leverage a cloud storage service such as S3. At a higher level of abstraction, services such as Twitter or Facebook have not only attracted millions of users to their main websites, but have also enabled a myriad of popular applications that are layered on top of those services by leveraging the public APIs they provide.

An important challenge that arises from this layering is that the consistency semantics of these cloud services are almost always not clearly specified, with studies showing that in practice these services expose a number of consistency anomalies to applications [7]. Furthermore, even in the cases where precise specifications exist, it is difficult for programmers to reason about their impact, and this may lead to violations of application invariants that were meant to be preserved [2].

In this paper, we argue that it is possible to build a middleware layer mediating the access to cloud services in order to obtain fine-grained control over the consistency semantics that these services provide. The idea is that we can design a library that intercepts every call to the service or storage system running in the cloud, inserting relevant meta-data, calling the original API, and transforming the results that are obtained in a transparent way for the application. Through a combination of analyzing this meta-data and caching results that have been previously observed, this shim layer can then enforce fine-grained consistency guarantees.

In prior work, Bailis et al. [3] have proposed a similar approach, but with two main limitations compared to this work. First, their shim layer only provides a coarse-grained upgrade from eventual to causal consistency. In contrast, we allow programmers to turn on and off individual session guarantees, where different guarantees have been shown to be useful to different application scenarios [9]. Second, their work assumes the underlying ⟨key,value⟩ store is a NoSQL system with a read/write interface. Such an assumption simplifies the development of the shim layer, since (1) it gives the layer full access to the data stored in the system, and (2) it provides an interface with simple semantics.

In this work, we propose a shim layer that allows for a fine-grained control over the session guarantees that applications should perceive when accessing online services. These services typically enforce rate limits for operations issued by client applications. For guaranteeing that this limit is the same when using our shim layer, a single service operation should be executed for each application operation. Furthermore, our layer is not limited to using online storage services with a read/write interface, since it is designed to operate with services that offer a messaging interface such as online social networks. The combination of these three requirements raises interesting challenges from the perspective of the algorithms that our shim layer implements, e.g., to handle the fact that online social networks only return a subset of recent messages, which raises the question of whether a message does not appear because of a lack of a session guarantee or because of being truncated out of the list of recent messages.

We implemented our shim layer and integrated it with the Facebook API and the Redis storage system. Our evaluation shows that our layer allows for fine-grained consistency upgrades at a modest latency overhead.

The remainder of this paper is organized as follows. Section II discusses our target systems and the assumptions made regarding the centralized system over which third-party applications are developed. Section III discusses the architecture and high level view of our system, while Section IV details the algorithms employed in our solution to enforce each of the session guarantees. Section V discusses our prototype implementation and presents experimental results obtained over two different services. Finally, Section VI discusses relevant related work and Section VII concludes the paper with some final remarks.

978-1-5386-1679-6/17 $31.00 © 2017 IEEE
DOI 10.1109/SRDS.2017.9

II. Target systems

Our goal is to provide particular consistency guarantees to third-party applications using popular online web services that expose public APIs. In particular, the application developer may choose to have individual session guarantees (read your write, monotonic reads, monotonic writes, and writes follows reads) as well as combinations of these properties (in particular, all four session guarantees corresponds to causality [5]). To achieve this, we provide a library that can be easily attached to the third-party client application, allowing us to enrich the semantics exposed through the system public API. There are multiple popular systems that provide such public APIs, with various differences in terms of the interface they expose. As such, we needed to focus on a group of APIs with a similar service interface that we can easily adapt to, and we chose to focus on a particular class of services, namely social networks, such as Facebook, Twitter, or Instagram. Our choice is based on the relevance and popularity of these services and also on the large number of third-party applications that are developed for them. In particular, we target services that expose a data model based on key-value stores, where data objects can be accessed through a key, and that associate a list of objects to each key. We observe that this data model is prevalent in online social network services, particularly since they share concepts such as user feeds and comment lists. In particular, we target services where the API provides two fundamental operations to manipulate the list of objects associated with a given key: an insert operation to append a new object to the first position of the list, and a get operation that exposes the first N elements of the list (i.e., the most recent N elements).

Since we access these services through their public APIs, we need to view the service implementation as a black box, meaning that no assumptions are made regarding their internal operation. Furthermore, we design our protocols without making any assumption regarding the consistency guarantees provided through the public service API. The importance of not assuming any guarantees from existing services is justified by our own previous measurement study [7], which showed a high prevalence of violations of multiple session guarantees in public APIs provided by services of this class.

Our algorithms require storing meta-data alongside the data, which can be difficult to do when accessing services as black boxes, namely when the service has no support for including user managed meta-data (this is the case of Facebook, which we explore in the context of our prototype experimental evaluation). In this case, we need to encode this meta-data as part of the data itself. As a consequence, when the service is accessed by native clients (i.e., web applications or third party applications that do not resort to our Middleware) the user might see this meta-data. However, we believe that this is not a crucial issue, since many third party applications only access lists that are used exclusively by that application.

In order to arbitrate an ordering among operations issued by the local client and other remote clients, our Middleware has the need to have an approximate estimate of the current time.

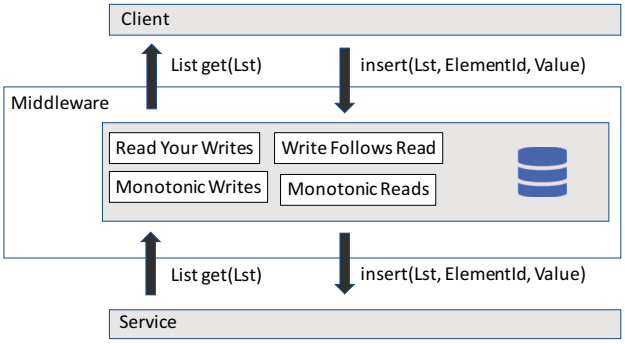


Fig. 1. Middleware

To achieve this, two options are available. If the service has a specific call in its public API that exposes the time in the server, such call can directly be used by our system. Otherwise, if the service exposes a REST API (which is typical in many services) a simple REST call can be performed to the service, and the server time can be extracted from a standard HTTP response header (called *Date*). Note that, even though it is desirable that this estimate is synchronized across clients, we do not require either clock or clock rate synchronization for correctness. In particular, the only negative effect of clocks being out of synch is a reordering of concurrent events from different sessions that is incoherent with their real time occurrence; this can imply, in the case of a service that outputs a sliding window of recent events, that more recent messages may be considered eligible for being truncated (i.e., considered older than the lower end of the window). However, we guarantee that such ordering never violates the correctness conditions we are enforcing.

Finally, we observe that, in practice, the public API exposed by these services often imposes rate limits for operations issued by client applications. These rate limits are exposed under the form of a maximum number of operations that can be executed within a given time window. In particular, we have experimentally observed that violating these rate limits can lead the service to either block further access by the application, or introduce noticeable delays in processing requests issued by the application. The existence of operation rate limits imposes a requirement on our protocols: for each application operation, a single service operation can be issued. This is important to guarantee that an application using our middleware faces the same rate limits as an application using directly the service.

III. System overview

In this section we discuss the general architecture of our solution, which is materialized in a library implementing a middleware layer. We then provide an overview of the operation of our protocols, explaining how they enforce the consistency guarantees of session properties in a transparent way for the client applications.

A. Architecture

Our system consists of a thin layer that runs on the client side and intercepts every call made by the third-party client application on the service, mediating access to the service. In particular, our layer is responsible for contacting the service on behalf of the client application, process the responses returned by the service and generate responses to the client applications according with the session guarantees being enforced. Figure 1 provides a simple representation of this architecture.

Our system can be configured by the third-party application developer to enforce any combination of the individual session guarantees (as defined by Terry et. al [9]), namely: i) read your writes, ii) monotonic reads, iii) monotonic writes, and iv) writes follows reads. In order to enforce these guarantees, our system is required to maintain information regarding previous operations executed by the client application, namely previous writes that were issued or previous values that were observed by the client. In addition, our layer can also insert meta-data that is stored alongside the data in the original system, but stripped by the library before the final response is conveyed to the client.

B. Overview

As mentioned, our system intercepts each request performed by the client application, executes the request in the service, and then processes the answer generated by the service to provide a (potentially different) answer to the client application. This answer is computed based on a combination of the internal state that records the previous operations that were run by that particular client, and the actual response that was returned by the service.

Tracking application activity. In order to keep track of user activity, our system maintains in memory a set of data structures for each part of the service state that is accessed by the application. These data structures are updated according both to the activity of the applications (i.e., the operations that were invoked) and the state that is returned by the service. These data structures are: i) the *insertSet*, which stores the elements inserted by the client and ii) the *localView*, which stores the elements returned to the client.

Enforcing session guarantees. Enforcing session guarantees entails achieving two complementary aspects. First, and depending on the session guarantees being enforced, some additional meta-data must be added when inserting operations. As mentioned, this meta-data can be either added to a specialized meta-data field (if the API exposed by the service allows this) or directly encoded within the body of the element being added to the list. Such meta-data has to be extracted by our library when retrieving the elements of a list, thus ensuring transparency towards client applications. Second, our system might be required to either remove or add elements to the list that is returned by the service when the application issues an operation to obtain the current service state, in order to ensure that the intended session guarantees are not violated.

In the next section, we discuss the concrete algorithms executed by our system upon receiving an insert or get operation for a particular list, in order to ensure that the values observed by the client application adhere to the semantics of each of the session guarantees that are intended to be provided.

Algorithm 1: Initialization of local state

```
1: upon init(Lst) do
2:     lstState ← init()
3:     lstState.insertSet ← {}
4:     lstState.localView ← {}
5:     lstState.lastTimestamp ← 0
6:     lstState.insertCounter ← 0
7:     listStates[Lst] ← lstState
```

IV. Algorithms

We now discuss in more detail the algorithms that are employed by our Middleware layer to enforce session guarantees, and the rationale for their design. To this end, we briefly remind what each of the four session guarantees entails (we extend the definitions previously introduced in [7]), and then explain why our algorithms ensure that the anomalies associated with each of the session guarantees are prevented by it.

We explain our algorithms assuming that the service offers an interface with the following two functions, which are in practice easily mapped to functions that are supported by the various services that we analyzed: the insertion of an element in a given list Lst, denoted by the execution of function `insert`(Lst,$ElementID$,$Value$), where Lst identifies the list being accessed, $ElementID$ denotes the identifier of the element being added (which can be an identifier generated by the centralized service or a unique identifier generated by our Middleware), and $Value$ stands for the value of the element being added to the list; and the access to the contents of a list, denoted by the execution of function `get`(Lst), where Lst identifies the list being read by the client.

When the client accesses a list Lst for the first time, a special initialization procedure is triggered internally by our Middleware (Alg. 1), which initializes the local state regarding the accesses to Lst. The initialization is straightforward: it creates the object *lstState* that maintains all relevant information to manage the accesses to Lst (line 2). This state is composed by the sets **insertSet** and **localView** that were discussed previously, and that are initially empty (lines $3-4$). Furthermore, two other variables are initialized, **lastTimestamp**, which is used to maintain information regarding elements that were removed from the previously discussed sets, and **insertCounter**, which tracks the number of inserts performed by the local client in the context of the current session. Both of these variables have an initial value of zero (lines $5-6$). Finally, the *lstState* variable is stored in a local map, associated to the list Lst (line 7). Next, we explain how this local state is leveraged by our algorithms to enforce the various session guarantees.

A. Read Your Writes

The Read Your Writes (RYW) session guarantee requires that, in a session, any read observes all writes previously

Algorithm 2: Read Your Writes

```
1:  function insert(Lst, ElementId, Value) do
2:      lstState ← listStates[Lst]
3:      Element e ← init()
4:      e.v ← Value
5:      e.id ← ElementId
6:      e.timestamp ← obtainServiceTimeStamp()
7:      SERVICE.insert(Lst, ElementId, e)
8:      lstState.insertSet ← e ∪ lstState.insertSet

9:  function get(Lst) do
10:     lstState ← listStates[Lst]
11:     sl ← SERVICE.get(Lst)
12:     sl ← orderByTimestamp(sl)
13:     sl ← addMissingElementsToSL(sl, lstState.insertSet, lstState.lastTimestamp)
14:     sl ← purgeOldElementFromSL(sl, lstState.insertSet, lstState.lastTimestamp)
15:     lstState.lastTimestamp ← getLastTimestamp(sl)
16:     return removeMetadata(subList(sl, 0, N))
```

Algorithm 3: Monotonic reads

```
1:  function insert(Lst, ElementID, Value) do
2:      Element e ← init()
3:      e.id ← ElementID
4:      e.v ← Value
5:      SERVICE.insert(Lst, ElementID, e)

6:  function get(Lst) do
7:      lstState ← listStates[Lst]
8:      sl ← SERVICE.get(Lst)
9:      lstState.localView ← appendNewElementsToTop(sl, lstState.localView)
10:     return removeMetadata(subList(lstState.localView, 0, N))
```

executed by the same client. More precisely, for every set of insert operations W made by a client c over a list L in a given session, and set S of elements from list L returned by a subsequent get operation of c over L, we say that RYW is violated if and only if $\exists\, x \in W : x \notin S$.

This definition, however, does not consider the case where only the N most recent elements of a list are returned by a get operation. In this case, some writes of a given client may not be present in the result if more than N other insert operations have been performed (by client c or any other client). Considering that the list must hold the most recent writes, a RYW anomaly happens when a get operation returns an older write performed by the client but misses a more recent one. More formally, given two writes x, y over list L executed in the same client session, where x was executed before y, an anomaly of RYW happens in a get that returns S when $\exists\, x, y \in W : x \prec y \wedge y \notin S \wedge x \in S$.

Alg. 2 presents our algorithm for providing RYW. To avoid the anomaly described above, the idea is to store, locally at the client, all elements that are inserted by the local client in the list and add them to the result of get operations. In the insert operation, the inserted element is stored locally by the client (line 8). Additionally, our algorithm stores some meta-data in the object before performing the insert operation over the centralized service (lines $5-6$). This information represents, respectively, the identifier of the element and a timestamp for the insert operation (from the perspective of the service, and retrieved as described in Section II). The element identifier is used to uniquely identify the writes. The timestamp and element identifier allow for totally ordering all entries in the **insertSet**, with the order being approximately that of the real-time order of execution. Note that the operation in line 12 also checks if the timestamps retrieved from the service in the same session are monotonically increasing, and, if not, enforces that property by overwriting the returned timestamp with an increment of the most recent one; this is important to avoid reordering events from the same session in case the timestamp provided by the server does not increase monotonically for some reason.

For executing a get operation (line 9) our algorithm starts by executing the get operation over the service (line 11). Then, the returned list (*sl*) is ordered (line 12) and all elements of the local **insertSet** that are missing in the list are added to the list, keeping it ordered (line 13). Before returning the most recent N elements (with no meta-data) (line 16), our algorithm removes old session elements from the *sl* list and updates the **lastTimestamp** variable with the timestamp of the oldest element of the client session returned to the client (lines $14-16$).

A limitation of this algorithm is that it causes the **insertset** to grow indefinitely. To avoid this, we use the timestamp of each element to remove from the **insertset** any element older than **lastTimestamp**. We also need to include the session id in the metadata of each element to avoid old elements of the session to reaper. We omit this from Alg. 2 for readability.

B. Monotonic Reads

This session guarantee requires that all writes reflected in a read are also reflected in all subsequent reads performed by the same client. To define this in our scenario where a truncated list of N recent elements is returned, we say that Monotonic Reads (MR) is violated when a client c issues two read operations that return sequences S_1 and S_2 (in that order) and the following property holds: $\exists\, x, y \in S_1 : x \prec y \textit{ in } S_1 \wedge y \notin S_2 \wedge x \in S_2$, where $x \prec y$ means that element x appears in S_1 before y.

To avoid this anomaly, our algorithm (presented in Alg. 3) resorts to the **localView** variable to maintain information regarding the elements (and their respective order) observed by the client in previous get operations. Therefore, when the client issues a get operation, our Middleware issues the get command over the centralized service (line 8) and then updates the contents of its **localView** with any elements that are returned by the service and that were not yet within the **localView** (line 9). These new elements are appended to the start of the list, as they are assumed to be more recent than those of the current **localView**.

The algorithm terminates by returning to the client the N most recent elements in the **localView**. These elements are exposed to the client without any of the meta-data added by our algorithms (line 10). Note that in this case the insert operation only issues the corresponding insert command with additional meta-data on the centralized service (lines $1-5$).

Similar to the previously discussed algorithm, this approach has a limitation regarding the growth of local state, in this

Algorithm 4: Monotonic Writes

```
1:  function insert(Lst, ElementID, Value) do
2:      lstState ← listStates[Lst]
3:      Element e ← init()
4:      e.id ← ElementID
5:      e.v ← Value
6:      e.clientSession ← getClientSessionID()
7:      e.sessionCounter ← lstState.insertCounter++
8:      SERVICE.insert(Lst, ElementID, e)

9:  function get(Lst) do
10:     lstState ← listStates[Lst]
11:     sl ← SERVICE.get(Lst)
12:     sl ← sortElementsBySessionCounters(sl)
13:     sl ← removeElementsWithMissingDependencies(sl)
14:     return removeMetadata(sl)
```

Algorithm 5: Write Follows Read

```
1:  function insert(Lst, ElementID, Value) do
2:      lstState ← listStates[Lst]
3:      Element e ← init()
4:      e.id ← ElementID
5:      e.v ← Value
6:      e.cutTimestamp ← obtainCutTimestamp(lstState.localView)
7:      e.dependencies ← projectElementIdentifiers(lstState.localView)
8:      e.timestamp ← obtainIncreasingServiceTimeStamp(lstState.localView)
9:      SERVICE.insert(Lst, ElementID, e)

10: function get(Lst) do
11:     lstState ← listStates[Lst]
12:     sl ← SERVICE.get(Lst)
13:     sl ← removeElementsWithMissingDependencies(sl)
14:     cutTimestamp ← highestCutTimestamp(sl)
15:     sl ← removeElementsBelowCutTimestamp(sl, cutTimestamp)
16:     lstState.localView ← appendNewElementsByTimestamp(ls, lstState.localView)
17:     lstState.localView ← purgeOldElements(lstState.localView)
18:     return removeMetadata(sl)
```

case the **localView** can grow indefinitely. To avoid this, we associate with each element inserted in the list a timestamp. This timestamp allows us to remove from the **localView** any element with a timestamp smaller than the timestamp of the oldest element that was in the last return to the client. We omit this from Alg. 3 for readability.

C. Monotonic Writes

This session guarantee requires that writes issued by a given client are observed in the order in which they were issued by all clients. More precisely, if W is a sequence of write operations issued by client c up to a given instant, and S is a sequence of write operations returned in a read operation by any client, a Monotonic Writes (MW) anomaly happens when the following property holds, where $W(x) \prec W(y)$ denotes x precedes y in sequence W: $\exists x, y \in W : W(x) \prec W(y) \wedge y \in S \wedge (x \notin S \vee S(y) \prec S(x))$.

However, this definition needs to be adapted for the case where only N elements of a list are returned by a get operation. In this case, some session sequences may be incomplete, because older elements of the sequence may be left out of the truncated list of N returned elements. Thus, we consider that older elements are eligible to be dropped from the output, provided that we ensure that there are no gaps in the session subsequences and that the write order is respected, before returning to the client. Formally, we can redefine MW anomalies as follows, given a sequence of writes W in the same session, and a sequence S returned by a read: $(\exists x, y, z \in W : W(x) \prec W(y) \prec W(z) \wedge x \in S \wedge y \notin S \wedge z \in S) \vee (\exists x, y \in W : W(x) \prec W(y) \wedge S(y) \prec S(x))$.

Alg. 4 presents the algorithm employed by our Middleware to enforce the MW session guarantee. We avoid the anomaly described above by adding meta-data to each insert operation (lines $1-8$) in the form of a unique client session id (*clientSession* – line 6) and a counter (local to each client and session) that grows monotonically (*sessionCounter* – line 7). This information allows us to establish a total order of inserts for each client session.

This meta-data is then leveraged during the execution of a get operation (lines $9-14$) in the following way. After reading the current list from the service (line 11), we simply order the elements in the read list (*sl*) to ensure that all elements respect the partial orders for each client session (line 12). Finally, an additional step is required to ensure that no element is missing in any of these partial orders. To ensure this, whenever a gap is found within the elements of a given client session, we remove all elements whose *sessionCounter* is above the one of any of the missing elements.

The get operation returns the contents that are left in the list *sl* without the meta-data added by our algorithms (line 14). Note that in this case we might return to the client a list of elements with a size below N. We could mitigate this behavior by resorting to the contents of the **localView** as we did in the algorithm to enforce MR. However, we decided to provide the minimal behavior to enforce each of the session guarantees in isolation.

D. Write Follows Read

This session guarantee requires that the effects of a write observed in a read by a given client always precede the writes that the same client subsequently performs. (Note that although this anomaly has been used to exemplify causality violations [1], [8], any of the previous anomalies represent a different form of a causality violation [9].) To formalize this definition, and considering that the service only returns at most N elements in a list, if S_1 is a sequence returned by a read invoked by client c, w a write performed by c after observing S_1, and S_2 is a sequence returned by a read issued by any client in the system; a violation of the Write Follows Read (WFR) anomaly happens when: $w \in S_2 \wedge \exists x, y \in S_1 : x \prec y$ *in* $S_1 \wedge y \notin S_2 \wedge x \in S_2$.

Our algorithm to enforce this session guarantee is depicted in Alg. 5. The key idea to avoid this anomaly is to associate with each insert the direct list of dependencies of that insert, i.e, all elements previously observed by the client performing the insert (line 7). Evidently, this solution is not practical, since this list could easily grow to include all previous inserts performed during the lifetime of the system. To overcome this limitation, we associate with each insert a timestamp based on the clock of the service, but with the restriction of

being strictly greater than the timestamp of any of its direct dependencies (line 8). Furthermore, we also associate with each insert a cut timestamp, that defines the timestamp of its last explicit dependency, i.e, the dependencies registered in the dependency list (line 6). The cut timestamp implicitly defines every element with a lower timestamp to be a dependency of that insert operation. By combining these different techniques, we ensure that the explicit dependency list associated with an insert has at most a value around N elements (which is the size of the **localView** maintained by our Middleware).

Since only N elements of a list are returned by a get operation, the older dependencies may be left out of the sequence that is returned. When this happens, it is safe to consider that these dependencies were dropped from the window that is returned, provided that we ensure that, for each element that is returned, all dependencies that are more recent than the oldest element are also returned.

In the get operation we leverage this meta-data to do the following: we start by reading the contents of the list from the service (line 12) and then over this list we remove any insert whose dependencies are missing. Thus, we only remove inserts whose missing dependencies have a timestamp above the insert cut timestamp. We then compute a cut timestamp for the obtained list *sl* (line 13) that is the highest cut timestamp among all elements in *sl*. We use this timestamp to remove from *sl* any element whose creation timestamp falls below the computed cut timestamp. Finally, before returning to the client the elements that remain in *sl* without the additional meta-data (line 18) we update and garbage collect old entries from the **localView** (lines $16-17$).

Similarly to the previous algorithm, the service might return a number of elements that is lower than N. In this case, to ensure that we always return N elements, we need to obtain the missing dependencies using a get operation that returns a single element (if supported by the service). In our implementation, we avoided this solution because it is prone to triggering a violation of the API rate limits. Again, an alternative way to address this is by, after reading the list from the service, merging its contents with those in the **localStore** and enforcing an order that is compatible with the timestamp of each element. However, for simplicity in exposition, we omit the details of this alternative.

E. Combining multiple session guarantees

Considering the algorithms to enforce each of the session guarantees discussed above, we can now summarize how to combine them. In a nutshell, it suffices for our Middleware to, on insert operations, add the meta-data used by each of the individual algorithms according to the guarantees configured by the application developer. Correspondingly, upon the execution of a get operation, our Middleware must perform the transformations over the list obtained from the service (*sl*) prescribed by each of the individual algorithms. Furthermore, all meta-data added to each element must also be removed before exposing data to the client application.

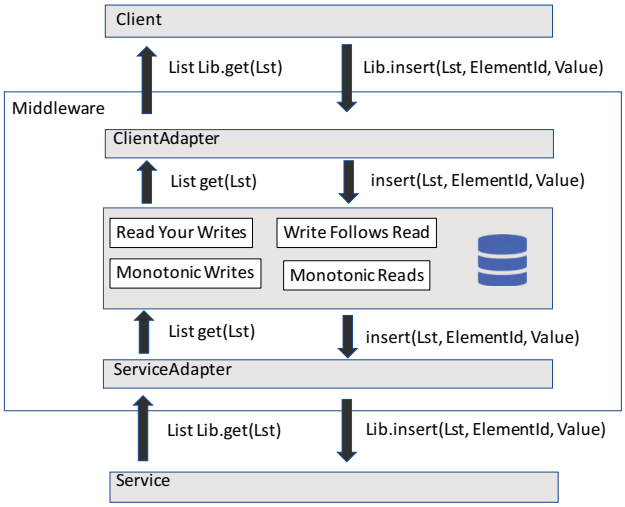


Fig. 2. Middleware with adapters

V. Evaluation

In this section we present the experimental evaluation of our Middleware, which compares the client-perceived performance obtained when using our Middleware to provide each of the session guarantees in isolation and their combination (i.e, enforcing all four session guarantees). In our experiments we used a prototype of our Middleware whose implementation we briefly discuss below. Our evaluation was made using two different geo-replicated online services. First, to illustrate the benefit of our Middleware when designing third-party applications that interact with online social networks we have used Facebook's public API. Then, to illustrate the operation of our Middleware when interacting with a service that imposes fewer restrictions on the number and timing of client operations, we experimented with a geo-replicated deployment of the Redis datastore managed by ourselves.

Our evaluation focuses on asserting the overhead that results from the use of our middleware, in terms of client perceived latency (for insert and get operations), the communication overhead due to the inclusion of additional meta-data, and the storage overhead, namely due to the need for our Middleware to locally maintain some information about previous operations performed by the client.

A. Implementation

Our prototype of the Middleware layer proposed in the paper was implemented in the Java language. To interact with the two services that we explore in this work, we resorted to the *restFB* library for Facebook[1], and the Jedis library for interacting with Redis[2].

Since our prototype was designed to interact with any Internet service with a public API, it requires two adapter layers to be written and provided to its runtime upon execution

[1]http://restfb.com
[2]https://github.com/xetorthio/jedis

(see Figure 2). These layers capture the API calls performed by the client application and translate them to a standard API exposed by our Middleware, and translate the calls to the centralized service performed by our Middleware into API calls to the library used to interact with the service, respectively. We have implemented these adapters for the two case studies employed in this evaluation. The adapters themselves are quite straightforward to write, and we believe most developers will be able to easily write new adapters to use our Middleware in combination with different libraries for accessing other online services.

B. Facebook Results

We have conducted our experiments with Facebook by using YCSB [6] to emulate clients using Facebook to post messages to a group feed and reading the contents of that group feed. To emulate such clients spread across the World, we run three independent YCSB instances in three different locations using Amazon EC2 instances in Oregon, Ireland, and Tokyo. Each YCSB instance uses 10 threads, emulating a total of 30 independent clients, for a total of 90 clients across the World. Each emulated client has an independent instance of our Middleware. To accommodate the rate limits of Facebook's public API, we impose a maximum of 15 requests per second per YCSB instance.

Each experiment reported in this section was executed 7 times, and different consistency guarantees were rotated along experiments, such that each different consistency guarantee had experiments running on different time periods of the day. This was done to remove experimental noise due to contention on the Facebook servers, e.g., due to other user activity. The workload executed by clients was a mix of 50% inserts and 50% gets. The Middleware was configured to have $N = 25$ meaning that each get retrieves at most 25 elements from the feed. Experiments reported in this section report the aggregated observations of $53,119$ insert and get operations.

1) Latency: We start by observing the latency of operations in Facebook when accessing the service directly through the library (labeled in the plots as NONE) and when using our Middleware to enforce each of the session guarantees in isolation and all of the session guarantees (labeled in the plots as ALL).

Figure 3 reports the latency observed for get operations, for all clients and per location of the client. Figure 3(a) shows that our Middleware introduces a small increase in the latency of get operations with a maximum increase of approximately one hundred milliseconds. Not surprisingly the overhead is at its maximum when all session guarantees are being enforced by our Middleware which is explained by a combination of the additional meta-data carried in each element, and the processing cost of the Middleware to perform the enforcement of each individual session guarantee.

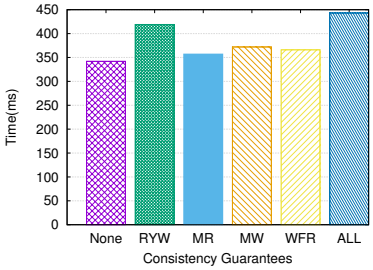
When observing the distribution of latency for requests according to the region where the client is located (Figure 3(b)), we note the same relative distribution in the results, with overall lower latency values for the clients in Oregon. This is explained by the latency of those clients towards the Facebook servers, which is notoriously smaller according to the latency when using the client library directly. Another noteworthy aspect of Figure 3(b) is that the observed latency has a visible variation, both across and even within different client locations. This suggests that the latency overhead in these cases may suffer from a noticeable variability due to external factors which are related with the architecture and deployment of such a large-scale real World application.

Figure 4 reports average latency results for the insert operation for all clients and per client location. The results reported in Figure 4(a) show that globally the latency penalty incurred by the use of our Middleware is again modest, with a maximum increase of at most 50 milliseconds. The individual session guarantee with the largest increase in latency is monotonic reads. Considering the latency values observed in different locations reported in Figure 4(b), we note the same pattern previously observed, where the latency experienced by clients in Oregon is lower compared with the remaining locations. This is expected, since this can be explained by the latency experienced by the client to contact the Facebook service in that concrete location when compared with the remaining locations used in our experimental work.

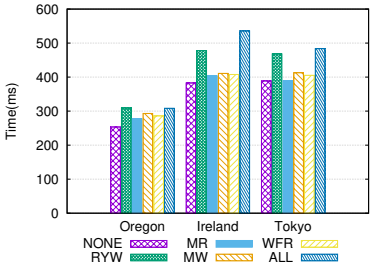
2) Communication Overhead: We now study the communication overhead imposed by our Middleware by observing the average size of messages exchanged between clients and the service. Figure 5 reports these results for each of the session guarantees and for their combination, compared with the use of the library without our Middleware, for both get and insert operations. The results show that the overhead introduced by our Middleware is low for the get operations. This happens because most of the payload in these messages are the multiple elements of the list that are returned. Since our algorithms use small meta-data objects, the communication cost remains dominated by the contents of the elements that are read, as can be observed in Figure 5(a).

The same is not true for insert operations, as reported in Figure 5(b). In this case, since each message contains only a single element to be added, the increase in message size is quite noticeable when the Middleware is enforcing Writes Follows Reads and the combination of all session guarantees. This happens due to the cost of sending the explicit dependencies of each inserted element, which can account to 25 unique element identifiers and their timestamps. The remaining session guarantees, in contrast, have a modest overhead of only a few tens of bytes.

3) Local Storage Size: Finally, Figure 6 reports the storage cost in terms of elements stored locally by our Middleware for enforcing each of the session guarantees and their combination. For completeness, we also provide the results for the NONE configuration, which, as expected, is zero. This is used as a sanity check for our results. Monotonic writes do not require any form of local storage, and therefore have no local storage overhead. In contrast, the remaining session guarantees resort to elements stored in the insertSet and localView data structures. As expected, when providing all of the session

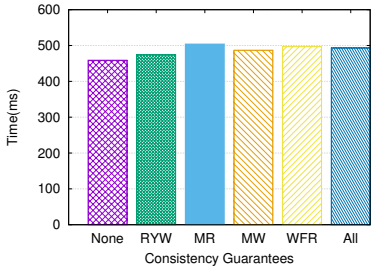


(a) Global

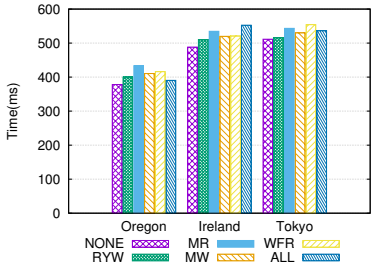


(b) Per location

Fig. 3. Latency of Get Operation in Facebook

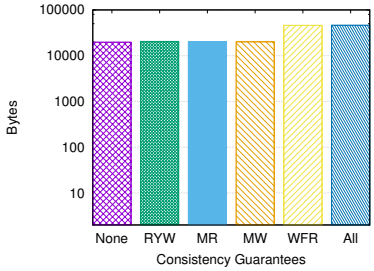


(a) Global

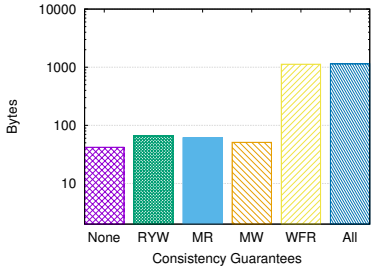


(b) Per location

Fig. 4. Latency of Insert Operation in Facebook



(a) Get Operation



(b) Insert Operation

Fig. 5. Communication overhead in Facebook

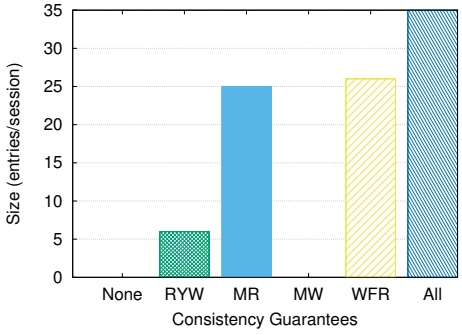


Fig. 6. Local storage overhead for Facebook

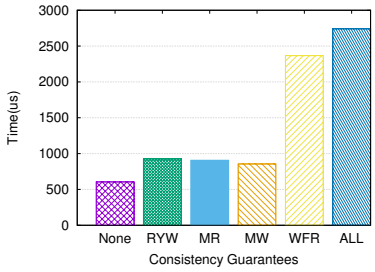
guarantees the local storage has more entries, this happens because the number of entries is the sum of the elements in the insertSet and in the localView.

C. Redis Results

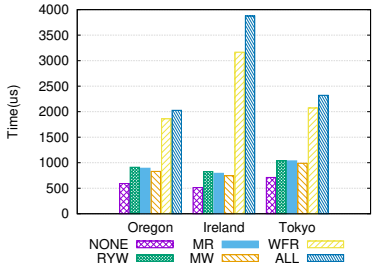
We also conducted experiments using the Redis data storage system. To this end, we have deployed Redis with its replication enabled across machines scattered in three Amazon EC2 regions: Oregon, Tokyo, and Ireland. Redis uses a master-slave replication model, and we have deployed the master in Ireland and two slaves in each region, for a total of 7 replicas. YCSB was executed in the same three regions of Amazon EC2 used in the previously reported experiments, with each YCSB instance running 10 threads that execute operation in a closed loop. Each thread has its own instance of the Middleware. All operations access the same list object stored in Redis, with the read operation being executed in one of the slave replicas, selected randomly. For each algorithm, we run our experiments 6 times for 60 seconds with an interval of four minutes between runs. Similar to the experiments conducted with Facebook, YCSB was configured to execute a workload composed of 50% inserts and 50% of reads. Again, we set N to be equal to 25. The experiments reported in this section aggregate the results from executing a total of $21,285,291$ insert and get operations.

1) Latency: Figure 7(a) presents the average latency of get operations. The results shows that our middleware introduces a very small overhead, on the order of microseconds, for Read Your Writes, Monotonic Reads, and Monotonic Writes. In Write Follows Read and when all session guarantees are enforced, there is an increase of approximately one to two milliseconds because the algorithms have to check the dependencies and process the meta-data. The results of Figure 7(b), which details the values observed in each region, show the same pattern across all regions, namely that the latency for reading data using a client in Ireland is higher than in other locations (due to the proximity to the master replica). This can be explained by the fact that writes in Ireland are much faster that in other locations, which causes the number of read operations that are executed to be higher in Ireland than in other locations, thus leading to a higher load, which results in a higher latency for executing operations.

In contrast to the experiments for the Facebook service, the observed latencies are much more predictable in this deployment. This confirms the expectation that a real-world service leads to qualitatively different results from a controlled experiment.

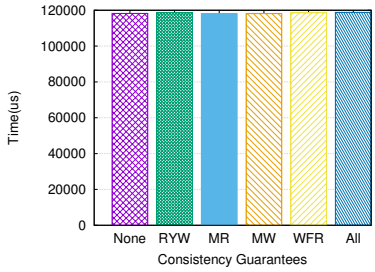


(a) Global

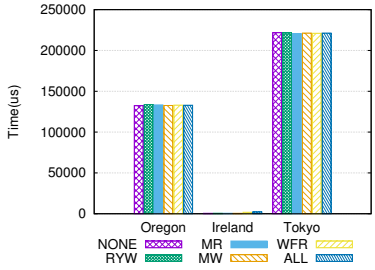


(b) Per location

Fig. 7. Latency of Get Operation in Redis

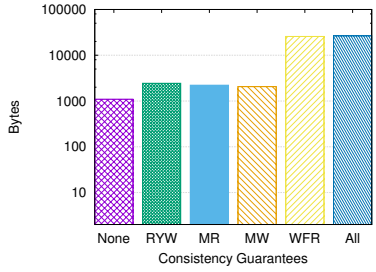


(a) Global

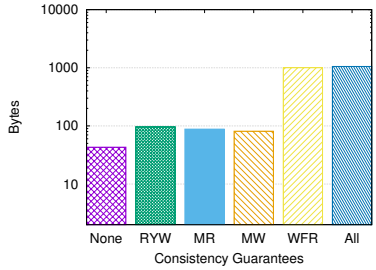


(b) Per Location

Fig. 8. Latency of Insert Operation in Redis



(a) Get Operation



(b) Insert Operation

Fig. 9. Communication overhead in Redis

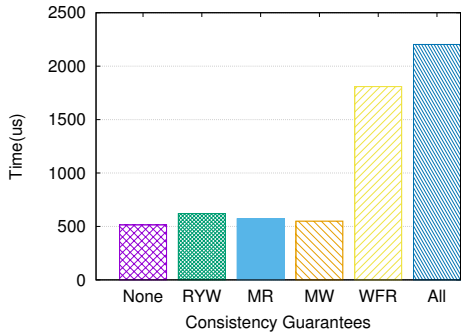


Fig. 10. Latency of Insert Operation in Redis in Ireland

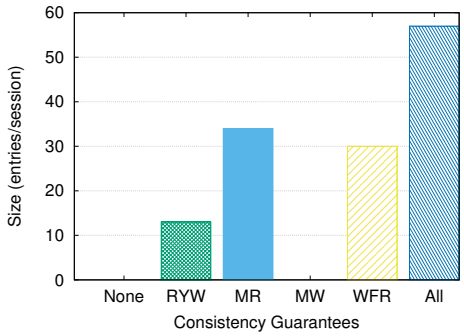


Fig. 11. Local storage overhead for Redis

Figure 8(a) shows the latencies of the insert operation, in this case the latency is almost the same across all cases, but if we look to Figure 8(b) we see that in Ireland latency values are much smaller, this is again justified by the location of the master replica in Ireland and the fact that all clients are issuing their write operations to the (same) master replica. Figure 10 reports the latencies in Ireland, again a similar pattern to the one observed for Get operations.

2) Communication Overhead: In terms of communication overhead imposed by our Middleware, the results in Figure 9(a) and Figure 9(b) show that in the Get and Insert operations the overhead is more noticeable when enforcing Write Follows Read and in when employing the combination of all algorithms. This happens due to the overhead associated with managing and communicating the information stored in dependency lists.

3) Local Storage Size: To conclude, Figure 11 shows that in Monotonic Reads and Write Follows Read the number of elements in the localView is around 30, which is higher than $N = 25$. This happens because of the high write throughput, which causes several elements to be assigned the same timestamp. In this case, our truncation algorithm allows for the limit to be exceeded in the case of ties. The combinations of all algorithms is also affected by this situation, leading to a higher value around 55. Note that we are showing the average of the highest value registered for each independent client session at any time during its execution.

VI. Related Work

The closest related work can be found in the recent proposals that also leverage a middleware layer that can mediate

access to a storage system in order to upgrade the respective consistency guarantees [3], [4].

In particular, Bailis et al. [3] proposed a system called "bolt-on causal consistency" to offer causal consistency on top of eventually consistent data stores. There are two main distinctions between bolt-on causal consistency and our proposal: first, we provide a fine-grained choice of which session guarantees the programmer intends the system to provide, and only pay a performance penalty that is associated with enforcing those guarantees. Second, they assume the underlying system offers a general read/write storage interface, which gives significant more flexibility in terms of the system design than in our proposal, which is restricted to the APIs provided by social networking services.

The other closely related system is the one proposed by Bermbach et al. [4], which is also based on a generic storage interface, namely that provided by S3, DynamoDB, or SimpleDB, in contrast to our focus on high level service APIs. While they also provide fine-grained session guarantees chosen by the programmer, they limit these to Monotonic Reads and Read Your Writes.

Our own prior work provides a measurement study of the violations of session guarantees that are observed when accessing real services [7]. However, the focus of that prior work is on understanding the prevalence of occurrences of lack of session guarantees, whereas this proposal is about fixing those problems through a middleware layer implementing a series of novel algorithms.

VII. Conclusions

In this paper we have shown that it is possible to enforce different consistency properties, in particular session guarantees for third party applications that access online services through their public APIs. We do so without explicit support from the service architecture, and without assuming that the service itself provides any of these guarantees. Our solution relies on a thin Middleware layer that executes on the client side, and intercepts all interactions of the client with the online service. We have presented different algorithms to enforce each of the well known session guarantees. Furthermore, our algorithms follow a simple structure that allows to combine then easily.

We have developed a prototype in Java that we used to evaluate our approach using two centralized services: Facebook, and a geo-replicated deployment of Redis. Our experiments show that we can enforce session guarantees with a modest overhead both in terms of user-perceived latency and communication with the centralized service.

Acknowledgements

This work was partially supported by the EU project LightKone (grant agreement n. 732505) and by FCT (UID/CEC/04516/2013). The research of R. Rodrigues is funded by the European Research Council (ERC-2012-StG-307732) and by FCT (UID/CEC/50021/2013). Part of the computing resources used for this work were supported by an AWS in Education Research Grant.

References

[1] Sérgio Almeida, João Leitão, and Luís Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, New York, NY, USA, 2015. ACM.

[3] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[4] David Bermbach, Jorn Kuhlenkamp, Bugra Derre, Markus Klems, and Stefan Tai. A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores. In *Proceedings of the 2013 IEEE International Conference on Cloud Engineering*, IC2E '13, pages 114–123, Washington, DC, USA, 2013. IEEE Computer Society.

[5] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. From session causality to causal consistency. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 152–158, Feb 2004.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[7] F. Freitas, J. Leitao, N. Preguia, and R. Rodrigues. Characterizing the Consistency of Online Services (Practical Experience Report). In *Proceedings of the 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 638–645. IEEE, June 2016.

[8] 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 Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[9] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.