# D6.1: Report on the first iteration of TaRDIS toolbox components

## Includes references to and descriptions of software prototypes.

Revision: v.1.1

| | |
|---|---|
| **Work package** | WP6 |
| **Task** | Task 6.1, Task 6.2, and Task 6.3 |
| **Due date** | 29/02/2024 |
| **Submission date** | 10/03/2024 |
| **Deliverable lead** | NOVA |
| **Version** | 1.1 |
| **Authors** | João Leitão (NOVA), Diogo Barreto (NOVA), Nuno Preguiça (NOVA), Miloš Simić (UNS), Diogo Jesus (NOVA), Diogo Paulico (NOVA), and Roland Kuhn (ACT) |
| **Reviewers** | Roland Kuhn (ACT), Panagiotis Trakadas (NKUA) |
| **Abstract** | This deliverable presents the progress of the TaRDIS project consortium on the design of novel distributed abstractions to support the operation of decentralised and swarm systems, focusing on abstractions for communication and membership primitives, distributed data management systems, and monitoring and management of decentralised systems. The report further presents and discussions software prototypes that were developed in the context of WP6 in the first year of the project. These prototypes will evolve to become components of the TaRDIS toolbox. |
| **Keywords** | Decentralised Membership Protocols, Decentralised Communication Protocols, Distributed Management Systems, Decentralised Monitoring, Decentralised Management |

www.project-tardis.eu

Grant Agreement No.: 101093006
Call: HORIZON-CL4-2022-DATA-01

Topic: HORIZON-CL4-2022-DATA-01-03
Type of action: HORIZON- RIA

## Document Revision History

| Version | Date | Description of change | List of contributor(s) |
|---|---|---|---|
| V0.1 | 03/01/2024 | Defined structure of the deliverable | João Leitão (NOVA) |
| V1.0 | 28/02/2024 | Complete draft of deliverable | João Leitão (NOVA), Diogo Barreto (NOVA), Nuno Preguiça (NOVA), Miloš Simić (UNS), Diogo Jesus (NOVA), Diogo Paulico (NOVA) |
| V1.1 | 09/03/2024 | Revision and content added | Roland Kuhn (ACT) |

## DISCLAIMER

**Funded by the European Union**

Funded by the European Union (TARDIS, 101093006). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

## COPYRIGHT NOTICE

| Project funded by the European Commission in the Horizon Europe Programme | | |
|---|---|---|
| **Nature of the deliverable:** | **R + DEM** | |
| **Dissemination Level** | | |
| **PU** | *Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page)* | ✓ |
| **SEN** | *Sensitive, limited under the conditions of the Grant Agreement* | |
| **Classified R-UE/ EU-R** | *EU RESTRICTED under the Commission Decision No2015/ 444* | |
| **Classified C-UE/ EU-C** | *EU CONFIDENTIAL under the Commission Decision No2015/ 444* | |
| **Classified S-UE/ EU-S** | *EU SECRET under the Commission Decision No2015/ 444* | |

* R: Document, report (excluding the periodic and final reports)

DEM: Demonstrator, pilot, prototype, plan designs

DEC: Websites, patents filing, press & media actions, videos, etc.

DATA: Data sets, microdata, etc.

DMP: Data management plan

ETHICS: Deliverables related to ethics issues.

SECURITY: Deliverables related to security issues

OTHER: Software, technical diagram, algorithms, models, etc.

# EXECUTIVE SUMMARY

This report presents the main results and outputs of work package 6 (WP6) of the TaRDIS project during the first 14 months of the project.

Work package 6 is responsible for developing and making available the fundamental technology that supports the operation and management of swarm applications. Due to this, WP6 is also a contributor to the TaRDIS toolbox, and produced prototypes of artefacts that can be integrated into the development of different swarm applications.

The activity of WP6 has been conducted across the three tasks that compose WP6, with the following main conceptual results and activities.

On task 6.1, which is focused on developing and making available membership and communication abstractions we have:

- Studied how to generalise the services and API of decentralised overlay networks and communication protocols into membership and other abstractions. We have shown the feasibility of using these abstractions by forking the Babel framework core. We have noted that supporting multiple interaction models did introduce visible overhead, and based on this decided to rely on simplified, and less flexible, versions of these APIs in the TaRDIS project.
- Based on the composition of a partial-view based membership decentralised service, and an epidemic-style broadcast protocols, we have proposed a novel decentralised global-membership abstraction. We have devised the protocol to ensure that if the membership of a swarm remains stable for a long enough period, and the network does not present partitions, then every node will observe the correct membership of the swarm.
- Conducted efforts to allow the integration of the Actyx middleware into the TaRDIS ecosystem, by extending its operation to include novel swarm communication primitives into its operation.

On task 6.2, which is focused on developing and making available distributed data management solutions for swarm applications we have:

- Proposed Arboreal, a novel distributed data management solution that extends from cloud infrastructures to the edge. Arboreal supports hundreds of different edge locations, and relies on dynamic fine-grained replication, where data objects are automatically replicated to edge locations where they are accessed (and removed when they are no longer useful). The system provided causal+ consistency guarantees, and this is achieved by leveraging on a hierarchical approach to interconnect data storage replicas that extends to the data placement scheme.
- Developed PotionDB, a geo-replicated storage system that provides strong eventual consistency under partial replication, where no location hosts the entirety of the data managed by the system. PotionDB provides a large-scale highly performant storage solution that allows swarm applications to interact with the closest datacenters only.
- Developed a set of adapters - for the Babel framework - that expose the API for interacting with data management systems defined by the TaRDIS consortium to interact with existing and legacy distributed data storage systems, including Cassandra, C3, Engage, and a blockchain-based system.

Finally, in the context of task 6.3, which is dedicated to providing the support to manage complex swarm systems, which include collecting telemetry and reconfigure components of such systems we have:

- Developed mechanisms to collect - locally at each process of a swarm – different performance indicators and telemetry information concerning the device where the process executes, the application process, and even protocol-specific indicators from protocols that are building blocks of the application. This information is collected in a centralised way at each process, which enables local correlation of telemetry information and performance indicators and can be exported in different formats.
- Explored how to take advantage of containerization technology, to reconfigure components of a swarm application by looking at these different components (i.e., processes) using hierarchical namespaces. We provide additional flexibility to the execution of these reconfigurations by enabling namespaces to be configured with labels, which allow reconfiguration targets to be defined based both on the hierarchy of namespaces, but also values of labels across components.

In terms of artefacts and software produced during the reported period, WP6 has made available several prototypes. We note that some of these prototypes have been written using the Babel framework that had been previously developed by members of the TaRDIS consortium, and whose goal is to simplify the development and execution of distributed systems through the composition of distributed protocols. The motivation for this is that Babel features an event-driven development API that is compatible (and well aligned) with the programming model being adopted by the TaRDIS consortium. The prototypes/reference implementations developed and made available are:

- An implementation and demonstrations of feasibility (using simple demonstrators) of the generic APIs for decentralised overlays and communication protocols as a fork of the Babel core.
- A simplified generic API for decentralised membership and communication protocols, that allow protocols developed using it to be easily exchanged among them when implementing swarm applications. This was achieved as a library for the Babel framework.
- We have two reusable decentralised and existing membership abstractions/protocols namely HyParView and X-BOT in the context of the Babel Framework. These implementations can work as reference implementations to both protocols, and they use the membership generic API described above.
- We have implemented two reusable decentralised and cooperative broadcast protocols that operate on top of membership abstractions detailed above. A flood-based broadcast and a gossip-based eager broadcast protocol.
- We have implemented several adapters for existing distributed storage solutions for the Babel framework, that expose the API for these components being considered by the TaRDIS consortium.
- We have implemented the prototype of the epidemic global view for the Babel framework. This prototype also serves as a reference implementation and is built by taking advantage (and composing) some of the components references in this list.
- A prototype (and reference implementation) for the Arboreal Cloud-Edge data management system.
- A prototype (and reference implementation) for the PotionBD geo-replicated storage system.
- An implementation of adaptors to existing storage solutions based on the API selected by the TaRDIS consortium. This includes support for the systems Cassandra, C3, Engage, and a blockchain system (IBM Hyperledger Fabric).

- A prototype of a distributed management and configuration system based on hierarchical namespaces.
- A fork of the Babel-core component that provides support for collecting and exporting runtime telemetry information about local protocol instances, applications, and the local device.

WP6 has produced four scientific papers and has disseminated the activities of the project in a tutorial held at an international conference.

In the next development cycle of TaRDIS, WP6 will address challenges related with security across two of its tasks and start to coalesce its results into tools that will integrate the TaRDIS toolbox, continuing to generate the runtime support for the development and execution of swarm applications.

## TABLE OF CONTENTS

# 1. Introduction

This report discusses the main activities of the Work Package 6 (WP6) of project TaRDIS in the first 14 months of the project. Whereas TaRDIS overall mission is to develop new solutions and technology that lower the expertise required to develop correct-by-design swarm applications. Swarm applications are applications that follow a mostly decentralised design and where application components exist in distinct, and potentially highly heterogeneous, devices that might operate under independent administrative domains, but that actively cooperate to achieve a common goal. WP6 is responsible for dealing with the inherent challenges of developing and making available fundamental building blocks and runtime support for such TaRDIS swarm applications.

The work package does so in three directions that have connections among themselves, and that map to the three tasks of the work package. The first task of WP6 (Task 6.1) is devoted to developing and making available decentralised abstractions to track the membership and support different communication paradigms in swarm systems. While there have been several contributions in the past regarding the design of scalable membership abstraction based on partial-views (i.e., where each element of the system is only aware of a small fraction of other elements of the system), understanding the trade-offs between these solutions, and verifying their applicability to different decentralised settings has not been systematically done in the past. Furthermore, from a software development point of view, typically these different solutions do not present a unified interface, that facilitates their interchange as components on concrete software pieces, which makes the reuse of (reference) implementations hard to achieve. Finally, in some concrete settings such as a smart factory, it is interesting to also support (eventually correct) scalable global membership services to support the operation of monitoring services that are not essential for the correctness of the system, such as to provide feedback - albeit imprecise - to humans or machines.

Complementary to the efforts in both developing and making available different solutions for membership management for swarm systems, Task 6.1 also explores different communication primitives for these settings, with emphasis on point-to-multipoint abstractions (e.g., broadcast, publish-subscribe). We note that there are several proposals in the literature for different solutions to these problems in decentralised systems, however, and similar to membership solutions, they lack a common interface that simplify the decoupling between a concrete solution and the application logic that uses it, and existing implementations are hard to reuse in different contexts. Task 6.1 addresses this by devising a common interface, making available different implementations that can be used across different applications, and exploring novel solutions that provide different trade-offs in the design space of such solutions. Notice that many of these communication primitives operate on top of decentralised membership abstractions, therefore these efforts are not disconnected from each other.

On top of these fundamental building blocks of swarm systems, the second task of WP6 (Task 6.2) focuses on devising and making available integration with distributed data management solutions. On one hand, this task is focused on devising solutions that, although are distributed , operate on dedicated infrastructure that can cover the space in the cloud-edge continuum. Such solutions differ among them on the abstractions and guarantees that are exposed to applications that use them on their design. This includes different data models, different querying capabilities, and maybe more importantly for several aspects of application logic, what are the consistency guarantees that are provided when considering that for availability and fault-tolerance, data must be replicated by these distributed storage systems. To further simplify the development of swarm applications, this task also has explored how to provide support for integration with existing, and well-known, distributed data management systems, such as Cassandra, and some of the systems previously designed by members of the TaRDIS consortium.

Related to this aspect, but in a much more disruptive research and development direction, Task 6.2 is also designing and implementing novel distributed data management solutions that do not require dedicated infrastructure for their operation. Instead, and in a much more natural fashion for swarm systems, these novel distributed data management systems emerge from the operation of different application components in the swarm. In this context there are many research challenges to be addressed, for instance ensuring data durability in this context is significantly more challenging. Maybe more importantly, in such a setting, the activity of a byzantine process in the system can disrupt the operation of the distributed data management system, which can have daunting effects on the application's correctness. These challenges are going to be addressed by Task 6.2 in the second half of the TaRDIS project.

Finally, the last vector being tackled by WP6 is focused on the monitoring and automatic control of swarm systems on its third task (Task 6.3). Due to the potential large scale and complexity of swarm systems, manual or human centric management can (easily) become unfeasible, and can easily become an error-prone task, with a non-negligible risk for the operation of swarm applications. To both address this problem and, in some cases, to improve the performance of complex swarm applications, Task 6.3 focuses on run-time support for autonomic management of these applications. This entails collecting telemetry information about the runtime operation of the system and resource consumption and performing some form of in-network processing and dissemination of this information towards locations that can make administrative decisions regarding the current system configuration. The second challenge to be addressed in this task, is how to coordinate the execution of reconfigurations of a system that might require administrative actions to be executed across many components, potentially scattered across different administrative domains.

Task 6.3 has the most interaction with the mechanisms for decentralised intelligence being developed in the project (which is part of the activities of work package 5), since devising a plan for reconfiguration of complex systems can easily become impossible through a set of rules, or even heuristics, defined by domain experts. Instead, it is much more interesting to take advantage of distributed machine learning mechanisms to empower the definition of reconfiguration plans, based on the acquired telemetry. Another challenge that must be tackled by Task 6.3 is on the fact that reconfiguration should avoid depending on a single centralised entity, and instead explore mechanisms, where different segments of the system have autonomy to perform reconfigurations, with minimal coordination with other segments, as to ensure scalability of these solutions.

The solutions and technology being developed and researched in the context of WP6 are made available, as much as possible, through open source tools and software, that also act as reference implementations for these solutions. Some of these artefacts (which we also discuss in this report) will be later integrated in prototypes of the TaRDIS use case or serve as the reference implementations for devising adequate components for those use cases.

The reminder of this report is organised as follows: Section 2 provides both the progress report on the activities of each of the tasks of WP6, in particular Section 2.2 reports on the main results of Task 6.1; Section 2.3 summarises the main results of Task 6.2, and Section 2.4 provides insights on the main results of Task 6.3; with Section 2.5 reporting on some of the immediate planned activities for these tasks. Section 3 provides pointers and brief descriptions or instructions on the software produced by WP6 that illustrates the results reported on Section 2.2. Section 4 provides a state of the art revision on the fundamental fields in which WP6 acts. Section 5 reports on complementary activities of WP6 in terms of scientific publications and dissemination activities; Section 6 briefly discusses point-of-contact and relationships with activities being conducted on the other technical work packages of TaRDIS; and finally, Section 7 concludes this document, with a summary of the main results achieved in the reported period.

# 2. Progress Report and Plan

## 2.1 Overview

In the reported period of TaRDIS (first 14 months) WP6 has focused on the following activities, which we further detail in the following text.

This deliverable, which is focused on Decentralised Membership and Communication Primitives, has explored generic APIs for both types of services, studying different solutions found in the literature to identify different types of services that these abstractions can provide. Based on this, we have developed within the context of the Babel [59] framework - which we are using in the project as a tool to develop prototypes and reference implementation - a simpler and pragmatic common API for this type of abstractions and implemented several solutions found in the literature both for partial-view based membership services (commonly referred as overlay networks) and application-level broadcast communication protocols, that are now provided as reusable (and interchangeable) components to implement swarm applications. To showcase the benefits of having such protocols as reusable components, we also designed and implemented a scalable global view membership solution that takes advantage of a partial-view membership protocol and a data dissemination communication primitive. We have also taken the existing proprietary Actyx middleware (owned by project partner ACT) and improved its module structure so that it can effectively and efficiently be consumed as an open-source building block; we have then open-sourced it and began work on reshaping it so that it can be used as a library instead of as a separate process.

Task 6.2 which is focused on decentralised data management and replication, has focused its efforts on developing novel distributed data management solutions that operate on dedicated infrastructure. As detailed further ahead in this document, these solutions are Arboreal, which is the first distributed data management system that can extend from cloud to the edge, featuring dynamic data replication and offering to application (so-called) *causal*+ consistency; and PotionDB a distributed storage solution system featuring strong eventual consistency while taking advantage of partial replication. In addition to these two main results, this task also developed and made available adaptors, that conform to a generic interface for interacting with data management system put forward by Task 6.2, for existing storage systems, including the well-known Cassandra distributed no-SQL database, and system proposed by members of the consortium in the past. The source code for these systems is provided also in the project repositories for ease of access. The Actyx middleware offers reliable and durable event stream replication, to which we have added data retention policies in anticipation of using Actyx as another swarm data storage system.

Task 6.3 which is focused on decentralised monitoring and reconfiguration focused, in the reported period, on developing mechanisms to systematically collect runtime telemetry from swarm application components, at different levels, including resource consumption at the device level, performance indicators from processes, individual distributed protocols, and applications. To do so, Task 6.3 developed a fork of the Babel framework core that provided the support for collection of metrics from different components of an application, and developed APIs for making this telemetry information available to the Babel core, allowing that information to be exported. The other direction in which Task 6.3 made progress in the reported period, was on devising a (currently centralised) solution for runtime reconfiguration of swarm application components based on namespaces and taking advantage of containerization technology to execute these reconfigurations, considering a design point relevant for legacy components of swarm applications that do not expose a reconfiguration API, an aspect in which Task 6.3 will work in the future.

## 2.2 Decentralised Membership and Communication Primitives (Task 6.1)

Nowadays, distributed architectures are widely used to build robust, scalable, and fault-tolerant systems. In distributed architectures, different components of the system - typically materialised by a process running on some machine - cooperate with each other over a network to perform a specific task or achieve some goal. In this context, a network, such as the Internet, is composed of a set of interconnected machines (or processes) in which information can be exchanged between pairs of these elements [18].

Systems that operate on top of a network can take advantage of a more centralised or decentralised approach. In a centralised system all information needs to be sent to a central point to be processed, thus making this central component responsible for a significant part of the operation.[1] Alternatively, in a decentralised architecture the processes present on the network can interact and cooperate directly without requiring an intermediary. This is achieved by having processes sending and receiving messages directly among themselves.

Steen and Tanenbaum discuss in [18] the difference between a distributed system and a decentralised system stating that both may rely on multiple machines performing a certain operation but in the first case “processes and resources are sufficiently spread across multiple computers”, e.g., an email service that relies on multiple servers as a way to distribute load and improve fault-tolerance, yet in the second case “processes and resources are necessarily spread across multiple computers” making the distribution of processes a core aspect of the system.

A decentralised approach offers many advantages when compared to a single process or a group of autonomous processes that cannot interact directly between them. These include improved availability by avoiding single points of failure since any process (in general) can replace any other process that fails during the execution of the system; better scalability by opening possibilities for clients and data to be distributed across nodes [19, 18], hence taking advantage of additional computational resources (across a potential large number of machines); improved reliability by replicating information across different machines (and across different geographic locations in some cases) minimising the risk of data loss; potentially decentralised systems can also increase privacy and robustness to malicious attacks due to the lack of a single target for setting up an attack and the possibility of using the nodes in the network to hide the identity of users or the exchanged information among them [20, 21].

On the other hand, centralised systems can be considered easier to maintain as information only needs to be sent to a central point [18]. Consequently, they do not have to deal neither with the heterogeneity questions related with the characteristics of each node nor the membership management and communication issues regarding the distributed nature of the infrastructure. In fact, when comparing centralised and decentralised systems it is not uncommon to consider decentralised systems algorithms to be more complex and difficult to understand and implement than their centralised counterparts.

While decentralised systems were popularised in the past by peer-to-peer systems [19], swarm systems can be perceived as an evolution of peer-to-peer architectures, that reap the benefits of having systems that are more flexible, cope well with heterogeneity, are more adaptable, and have improved robustness and scalability. Naturally, the foundations of the swarm

[1] The reader should note that the central component might only be logically centralised, in the sense that it might be materialised by several co-located machines, such as a service running on a data center.

systems share commonalities with peer-to-peer systems. In particular, underlying the operation of a swarm system, we require two fundamental abstractions: i) membership abstractions, that track at each moment which process are currently part of the system, providing to each element of the swarm information - even if incomplete - about other processes with whom it can cooperate and interact; and ii) communication abstractions, that provide to each process in the swarm the ability to exchange information with other elements of the swarm, potentially with different guarantees.

**Membership abstractions.** Ideally, any decentralised system would operate with access to the full membership of the system at each individual device, ensuring that each process knows about every other process in the system. Unfortunately, in dynamic settings, where processes can fail, join, or leave the system concurrently, the cost to maintain such information up to date becomes prohibitively high [22,23,24].

To address this challenge, a common solution is to allow nodes to only maintain information about a small set of other participants in the system, i.e., relying on a partial view of the system [22, 23]. In this case, each process in this system runs an instance of a distributed protocol that manages the contents of its own partial view, usually called the process neighbours, reflecting changes to the system membership. While there are different strategies to manage such partial views [22,25,26] the closure of the neighbouring relationships denoted by the partial views of each process denote an application-level network that is often called overlay network in the literature [22,25,26,27,28,19].

Overlay networks can have numerous connected processes (or devices) cooperating, for instance, to process large amounts of data or to share information in a decentralised way, as in the BitTorrent [29] protocol. There are scenarios where they operate at a much smaller scale, consider for instance applications of Internet Of Things (IoT) related with the management of sensors and actuators in a house exchanging information between themselves [30], or industrial machines on a factory exchanging data and coordinating their activities [31, 32, 33], decentralised solutions for the management of energy grids [27, 45, 6], or swarms of satellites in space communicating and autonomously adjusting their positions to avoid collisions [34]. Naturally, different scenarios may benefit significantly from different overlay networks (or membership abstractions), since there is no silver bullet solution that will be able to cope with different scales, workloads, application requirements, etc.

In particular, there are two main families of overlay networks that while materialising a membership abstraction based on a partial view, can provide additional functionalities for application components operating on top of them. Unstructured overlays, such as HyParView [22], Cyclon [25], Scamp [26] have random topologies (i.e., neighbouring relations between nodes are defined at random) and are good for applications that promote random interactions (for instance executing anti-entropy [38] between nodes to synchronise information, or execute gossip protocols to disseminate date [36,37]).[2] In contrast, structured overlays (popularised by distributed hash tables) such as Chord [39], Tapestry [40], Pastry [41], Kademlia [35], rely on global coordination strategy to manage the contents of partial views, which allows them to perform efficient application-level routing, and locate nodes (or resources on nodes), while being more expensive to manage and less fault tolerant than their unstructured counterparts [19].

[2] We note that there are also unstructured overlay solution [42,43] that while being random in nature, can adapt the topology of the overlay over time (in a decentralised manner) to optimise the overlay for some set of application-specific criteria, for instance, to promote low-latency links as part of the overlay to speedup the dissemination time of information executed on top of that overlay.

**Communication abstractions.** Like membership abstractions for decentralised systems (which, as discussed previously include both swarm and peer-to-peer systems) there are a plethora of possible communication abstractions that can be leveraged when building such a system. These can be decomposed in two main groups: point-to-point abstractions that enable the communication between two processes, and point-to-multipoint abstractions which provide a means for a process to send information to a set of other processes.

While point-to-point abstractions are simpler to implement, they can have different guarantees and information exposed to the application that can affect their operational cost. For instance, a point-to-point communication abstraction, might be best-effort and provide no feedback to the application regarding the delivery of information to a given destination, or by opposition, it could provide delivery guarantees in some scenarios and feedback back to the application when some information cannot be confirmed to be delivered successfully to the destination[3].

More interesting however are point-to-multipoint abstractions, since there are different classes of these abstractions that provide different API and promote different types of interactions between processes that form a swarm application. Point-to-Multipoint abstractions include application level and collaborative broadcast [22], multicast [104], and publish-subscribe [105], in turn the guarantees provided by these abstractions can be different, for instance all of them can operate in best-effort mode or be probabilistically reliable, allowing to explore different implementation alternatives in the design space that feature different operational costs and resource consumption.

When designing communication primitives - particularly the point-to-multipoint ones - for swarm systems, one typically requires the use of a (decentralised) membership abstraction over which a distributed communication protocol can be deployed that allows to implement the communication primitive. Consider for instance the broadcast primitive, that fundamentally allows a process to disseminate information to all participants of the system that are active around the time of the transmission. Such a primitive can be implemented on top of an unstructured overlay (which typically have a low maintenance cost and high fault-tolerance as discussed previously) using different alternatives, such as flooding messages over a random graph, which will ensure that all correct processes will receive a message as long as the underlying overlay network is connected [22] (i.e., there is at least one path between each process of the system and every other process). This however might yield too many redundant messages being transmitted and received by processes, which consumes not only bandwidth but also CPU. To lower this cost, we could instead rely on a gossip-based approach [44], where each node will select among its neighbours a configured amount of targets (called typically the *fanout* of the protocol) to retransmit each message received for the first time. This can lower the communication cost, but the guarantee that all processes receive a message that is broadcasted becomes probabilistic and dependent on the value of the fanout parameter. A different alternative is to take into consideration the feedback of the dissemination process of previous messages to select to which neighbours to transmit a new message, generating emergent structures [45,46] or even spanning trees in a fully decentralised manner [47,37]. While this tends to minimise the global bandwidth and CPU cost by lowering the number of redundant messages, such approaches might not be suitable for scenarios where the membership of the system changes frequently. Furthermore, such approaches can lead to unbalanced in the load imposed to individual nodes in the system, where a small fraction of processes are responsible for forwarding the largest fraction of messages, although there have been efforts in the past to devise solutions that mitigate this effect while also improving fault-

[3] Notice that while these abstractions have a relationship with transport protocols such as TCP, UDP, or QUIC, the concepts discussed here go beyond the layer 3 of the classical OSI model, since we are considering retransmissions and feedback mechanisms back to the application which are not part of the interface exposed by such protocols.

tolerance, by combining different interior-node disjoint emergent trees with network-coding [48].

**Task 6.1 Objectives.** A key insight to extract in relation to both membership and communication abstractions is that the design space for these primitives, in the context of decentralised systems in general and swarm systems in particular, is quite large.

Moreover, different designs explore trade-offs between guarantees, operational costs, and assumptions regarding the execution environment. Identifying the correct membership and communication abstractions and specific implementations for a given swarm application is therefore not a trivial task.

To further complicate this, many designs of these abstractions rely on specialised APIs, with the consequence that if a swarm application is developed on top of a particular abstraction, it will become hard to switch that abstraction by a different one in the future.

Considering this, Task 6.1 has the overall mission of designing and validating decentralised membership and communication primitives that can address the requirements of the different swarm applications, including the use cases and solutions developed across the TaRDIS project.

in more detail this task develops and validates:

- Underlying abstractions for supporting the development and efficient operation of higher-level data management (Task 6.2) and reconfiguration services (Task 6.3).
- Decentralised membership services that are responsible to maintain information about the active elements (e.g., devices/processes) in a swarm system. Such services can optionally authenticate participants in a system.
- Decentralised communication primitives that operate on top of the membership services to provide point-to-point and point-to-multipoint communication primitives with different guarantees (e.g., reliability, feedback to programmer) supporting different programming models (including support for publish/subscribe models, application-level multicast/broadcast).
- Provide a comprehensible suit of different abstractions that can be used as much as possible in an interchangeable fashion to develop swarm applications.

In the following we report on the results produced by Task 6.1 in the first year of the TaRDIS project:

- Section 2.2.1 reports on our initial efforts to devise generic APIs for decentralised overlay and communication primitives that decouple the logic of swarm applications from the concrete abstractions being employed to support the operation of the system.
- Section 2.2.2 reports on the initial design of a scalable global membership service that is built on top of a partial view membership abstraction and a decentralised broadcast primitive. Such a primitive can be used by swarm systems to collect and monitor the evolution of the membership of (medium scale) swarm systems, being of practical use for instance for the use case of a smart factory put forward by Actyx.
- Section 2.2.3 reports on the effort of getting the Actyx middleware ready for inclusion into the TaRDIS toolbox: we plan on extending and improving it with the swarm

communication primitives detailed above as well as using it as the communication medium underlying the workflow based TaRDIS communication model described in deliverable D3.1.

### 2.2.1 A Generic API for Decentralised Overlay and Communication Protocols

#### 2.2.1.1 Overview and Motivation

As discussed previously, decentralised and swarm applications typically rely on, or more precisely are built on top of, decentralised protocols, that provide abstractions and (distributed) services that simplify the design of the application.

These applications interact with decentralised protocols and their services through interfaces exposed by them. Many protocols have been proposed, in the context of peer-to-peer architectures, that provide, in different ways, abstractions related with membership management and support for different communication primitives between peers. Unfortunately, each protocol, even if it can be seen as offering a given type of service, typically exposes different interfaces, which leads application implementations to be fully entwined with the protocols used during their development, and additionally, makes it very hard to change the implementation to use a different protocol.

Considering the (extensive) literature in peer-to-peer systems, overlay networks can be used, as discussed previously, to provide a membership service. In fact, various examples of protocols that build and manage overlays networks exist, such as Chord [39], Kademlia [35], Freenet [20], HyParView [22], among others [50, 25, 51, 52, 37]. Examining these protocols it is easy to identify two classes of protocols based on the way peers are organised and connected between them leading to different network topologies: structured and unstructured overlay networks. The first ones are defined by enforcing pre-defined restrictions on the connections between peers, thus leading the overlay network to evolve towards a specific topology (e.g., nodes can form a ring, a tree, among others). The second ones do not enforce any type of network topology, allowing nodes to organise freely in a flexible way (usually randomly) [19, 42].

In addition to the lack of standard interfaces that are materialised by protocols offering an equivalent abstraction (e.g., membership, broadcast), there is the additional difficulty that a single protocol can actually offer different types of services as part of their operation, when we refine these abstractions beyond the high level functionalities of membership and communication. For instance, consider a distributed protocol such as Chord or Kademlia. As part of the operation of the protocol, it maintains (locally at each node) information about other active processes in the system, hence it provides a membership service - that ideally should be exposed by a common membership - but since these protocols are DHT, they are provide application-level routing services - which in turn should be exposed to applications though a common application-routing interface.

Therefore, the motivation for this activity comes from the need of identifying and defining new generic abstractions that can effectively support swarm applications, namely the ones related with membership management and communication, and their subtypes. These abstractions should be related with the services provided by protocols to applications, in opposition to the current protocol-dependent ones to decouple application implementations from concrete distributed protocol offering these abstractions as much as possible.

The main contributions that resulted from this activity were:

- A study of multiple decentralised protocols with the objective of identifying common functionalities/services provided by them, thus allowing the definition of a set of protocol-independent interfaces to expose that functionality.

- A proposed architecture that integrates and maps the defined protocol-independent interfaces to different distributed protocols, which allows decentralised and swarm applications to be developed focused on the abstractions they use instead of the concrete protocol that materialises those abstractions.

- We developed a reference implementation, based on a fork of the Babel framework (discussed in further ahead) in Java , of this proposed architecture. This reference implementation includes some well-known decentralised protocols implementations to illustrate the capabilities of the proposed architecture.

- A set of use-case simple applications that illustrate the benefits of using the protocol-independent interfaces that we have identified.

- An experimental evaluation that on one hand shows the decrease in complexity of implementing decentralised applications using our proposal and shows that our architecture introduces minimal overhead at runtime.

#### 2.2.1.2 Identifying Decentralised Abstractions and Interfaces

There is many decentralised protocols, many of which provide similar functionality but relying on different approaches more suitable for specific execution scenarios or operational conditions. We considered a set of such distributed protocols to identify common functionalities provided by them and derive protocol-independent interfaces for these functionalities (that we dub decentralised services)

As an example, consider the pairs of protocols Kademlia [35] and Chord [39], and HyParView [22] and Cyclon [25]. By comparing the first two it is possible to note that both protocols implement a functionality able to provide applications with the node (or the set of nodes) whose identifier(s) are closer to a given identifier. In the second case both HyParView and Cyclon are able to provide a sampling of peers that are part of the system. In both pairs of protocols we can derive a single interface that exposes these functionalities that are independent of the protocol, uniformizing the way applications can take advantage of that decentralised service.

Another challenge faced when building decentralised and swarm applications is related with the code complexity needed to instantiate and interact with underlying protocols, in particular when an application needs to rely on multiple protocols simultaneously. Decoupling applications from specific protocols requires that the selection of the concrete protocol that will materialise a given abstraction can be delayed until the application executed, ideally with that selection being independent of the application code (this can be useful for instance, if a new protocol appears that provides a service used by the application that is better suited for that application later).

This can be achieved by allowing applications to only define the decentralised services they require, the properties/guarantees of those services, and a set of parameters related with the instantiation and operation of protocols that can provide that service. These parameters can include, for instance, their own network address or the contact nodes to be used when joining the network.

##### *2.2.1.2.1 Generic decentralised Services Interfaces*

Following the design of Babel [59] that is being used in TaRDIS to implement prototypes of solutions derived in WP6, we assume an event based interface. When designing the interfaces for each service, we not only considered the operations that are made available to applications (through events of type Request, that can optionally generate events of type Reply), but also consider Notifications. Aligned with Babel, we define notifications as information that applications/protocols can receive (asynchronously) from (other) decentralised protocols, through each service interface, but that are not a direct result of a requested operation, e.g., a notification about a suspicion that some other process in the system has failed.

We also considered the existence of service-related properties in each interface, such as the type of overlay (structured or unstructured) in which a protocol, implementing the interface, relies on. As a result, a set of properties were defined for each service. These properties are optional and each protocol, providing a set of services and therefore implementing a set of interfaces, can then define which properties they have and their respective values. Service-related properties can be used to guide applications in choosing the most adequate protocol to materialise a decentralised service.

Before presenting protocol-independent interfaces identified by us, it is important to offer explanations for the type of interface we present and some parameters that are widely used in the description of their operations. In some operations we rely on parameters of the type Host. This type can be considered as the information required to identify and contact a given device or process in a swarm system, e.g., a pair IP:Port. Some operations also contain a parameter, called requestId, with a type byte array. This parameter makes it easier to use the asynchronous event-driven Request/Reply interface, by guaranteeing that the identifier provided on the request is returned on the reply, therefore allowing applications to easily match a reply with the respective request.

For this work we have considered most of the protocols referenced up to this point, and based on their analysis we have identified four distinct decentralised services - membership management, data dissemination, application-level routing, and resource storage - which we define more precisely in the following and for which we devised protocol-independent interfaces that we also present.

## Membership Management

The Membership Management service interface is related with the network management operations that are provided by decentralised protocols. We expect that almost all protocols need to implement this interface as they are required to provide a management service with operations for joining or leaving a network and getting either logical neighbours or samples of other processes in the system. This is the fundamental service (and interface) of protocols that provide a peer sampling service [54], like HyParView or Cyclon since these protocols were fundamentally designed to provide a partial view (stable or dynamic) to local peers. We now present the interface that we derived for this decentralised service divided into *Operations* (Requests, some of which might generate replies) and *Notifications*.

### *Operations*

**Join(Set<Host> contacts)** The Join operation is responsible for allowing a node to join a network given a set of process identifiers (defined here with the Host data type) are used as contact (a process that is already part of the system and that can be contacted to introduce the new participant into the system). The way each protocol takes advantage of this set for joining the network is protocol-dependent, e.g., a protocol can use only the first node on the set (eventually contacting other ones if some fail to reply), multiple nodes with no specific order, or multiple nodes by the order they appear on the set.

**Leave()** The Leave operation allows a node to leave the network. This operation does not require any parameters and protocols can either rely on an implementation that executes tasks to gracefully leave the system, minimising the impact on the system, for instance, by transferring information to other nodes; or an empty implementation will make that a node that leaves the network intentionally to be treated as if it has failed. A node that leaves the system through the leave operation should be able to join again latter using the Join operation.

**GetNeighborsSample(byte[] requestID, Integer t) → (byte[] requestID, Set<Host> sample)**
The GetNeighborsSample operation allows to obtain from the membership management service a set of process identifiers that are part of the system. The returned process identifiers depend on the logic of the protocol implementing this abstraction. The GetNeighborsSample operation takes as parameters the requestId and t. The requestId parameter is a unique identifier that identifies the request, and t is an optional Integer parameter allowing to define how many processes should be returned. This operation must generate a reply that contains the requestID and a set of process identifiers with a size of at most t, if t was defined in the request.

***Notifications***

**NeighborUp(Host h)** We considered the NeighborUp notification to be asynchronous triggered by a protocol when a new peer becomes known, i.e., when a new process can now be returned by the GetNeighborsSample operation, if no limitation on the set size is requested. The notification only has one field of type Host, containing the identifier of the new peer.

**NeighborDown(Host h)** We considered the NeighborDown notification to be triggered by protocols that implement the membership management service interface when a process in the system is detected as no longer being available, i.e., that process can no longer be in the set returned by the GetNeighborsSample operation. The notification only has one field, represented here with the Host data type, containing the information about the removed process.

***Properties***

A set of properties, related with the Membership Management service interface, was also defined in our solution. As explained before these properties are optional and assist applications in choosing the best protocol, considering their specific requirements, between all protocols providing the same service. Therefore, protocols may or may not provide values to those properties. The properties considered for this service are presented below.

**View Type** The View Type property defines the type of network view, Global or Partial, of a given protocol. If a protocol maintains a partial view of the network, a mechanism needs to exist to update it, more details on this are provided below when describing the Peer Sampling Type property.

**Peer Sampling Type** This property can be used by peer sampling protocols to define its peer sampling service as Static or Dynamic. We consider that a peer sampling protocol provides a static sampling service when the set of nodes returned by the GetNeighborSample operation does not change across executions of that operation if no nodes join or leave the system. Conversely, a peer sampling protocol can be considered as dynamic if the sample of nodes returned suffers changes over time, even if no nodes were added or removed from the network.

**Request Nodes** This property is closely related with the numNeighbors parameter of the GetNeighborsSample operation and is defined as a boolean value. If the property is set as true this means that the protocol supports the definition of a maximum number of nodes to be returned when performing the GetNeighbors operation, i.e., considers the numNeighbors parameter. If the property is set as false, the protocol does not consider the numNeighbors parameter.

**Overlay Structure** This property allows the classification of a protocol as structured or unstructured, based on the type of overlay network maintained by the protocol.

## Application-level Routing

The Routing service interface provides the operations related with the capability of obtaining a node, or a set of nodes, present on the network, based on the proximity of their logical identifier to some other identifier, therefore effectively allowing routing information to a particular node. DHT protocols, like Kademlia and Chord, provide this service by returning the nearest nodes to an identifier based on a protocol-dependent distance notion.

Protocols focused on resource location both based on exact-match queries, like Kademlia or Chord, and non-exact match queries, like Gnutella [55] or Gia [52], can materialise this interface.

### *Operations*

**FindNodes(byte[] requestID, byte[] searchData) → (byte[] requestID, Set<Host>)** The FindNodes operation is the foundation to route information to a given node or set of nodes. The operation receives two parameters, requestID and searchData. The requestID parameter, defined as a byte array, is used to match requests and corresponding replies, whereas the searchData parameter contains the query to be verified when performing the routing operation. With the objective of maintaining the operation as generic as possible, we do not make any assumption on the data type of queries that can be sent to protocols to perform the routing operation and, as so, we consider the searchData parameter as being a byte array. Thus, protocols implementing the operation are responsible for processing the query data received in accordance with their specific operation, e.g., protocols like Gnutella or Gia may consider the searchData to be a textual representation of a resource, whereas protocols like Kademlia or Chord can perform a hash on the data provided and use the result to return the nearest nodes based on some distance metric. The operation returns the requestId provided and a set of nodes (defined here with the Host data type) containing the result of the routing operation. The use of a set as a return value is explained by the necessity of maintaining the operation as generic as possible, to allow it to be implemented by different protocols designed independently and that operate under different assumptions.

### *Notifications*

We have identified no asynchronous notification that would make sense in this class of decentralised services.

### *Properties*

**Multiple Results** The Multiple Results property consists of a Boolean value specifying if a protocol can return multiple nodes when performing the FindNodes operation, i.e., if the returned set may contain more than one element, or otherwise, if the protocol returns at most one element.

## Resource Storage

The Resource Storage service interface provides operations that can be leveraged by protocols implementing mechanisms for resource storage and location, which is a common usage of Distributed Hash Tables (or DHTs). Many protocols that provide the Routing service, will also provide Resource Storage, however we consider that both services have significant differences regarding their operations and can be presented as two distinct services.

Therefore, while the Routing interface identifies a set of processes through a query, the Resource Storage operations are related with the ability of providing decentralised resource storage, independently of the underlying mechanisms.

***Operations***

**PutResource(byte[] key, byte[] data)** The PutResource operation allows storing a new resource on the system. Our interface defines two parameters for this operation: the key parameter, used as key for the resource (e.g., the name of the resource), and the data parameter, which contains the content. As we do not make any assumption about the key or data types, we consider both fields as being an array of bytes. Protocols implementing this operation should then store the resource in some node(s) present on the network allowing later retrieval. We note that this interface does not prescribe or exposes any replication strategy which is relevant for ensuring content availability in face of failures. Such mechanisms are implementation dependent.

**GetResource(byte[] requestID, byte[] key) → (byte[] requestID, Boolean found, byte[] key, byte[] data)** This operation allows the retrieval of a resource, stored on the decentralised system, by providing the respective key. We propose two parameters for performing this operation: a requestID parameter, defined as a byte array, to facilitate the use of this operation in asynchronous environments; and a key parameter, a byte array which contains the identifier of the resource to be retrieved. The operation should return the requestID, a boolean found indicating if the requested resource was obtained, the key identifying the resource that was queried, and the content (defined as a byte array named data), whose value only has meaning if the found field is set to true.

**RemoveResource(byte[] key)** The RemoveResource operation allows for the removal of a resource stored on the system, given its identifier. The operation only receives a key parameter, defined as a byte array, which represents the identifier of the resource to be removed.

***Notifications***

**NewResource(byte[] key)** The NewResource notification should be triggered within a particular process of the system when a resource is stored (or updated) at that node (at the protocol implementing the resource storage interface). The notification contains the key of the inserted resource.

**RemovedResource(byte[] key)** The RemovedResource notification should be triggered within a particular process when a resource is removed from that node. The notification contains the key of the removed resource.

## <u>Dissemination</u>

The Dissemination service interface supports a fundamental point-to-multipoint communication primitive. This interface should be implemented by protocols that are able to disseminate (e.g., broadcast or multicast typically in a collaborative way) a message throughout all participants of the system.

Protocols responsible for the dissemination of information, such as push-based gossip protocols [24], Plumtree [37], GoCast [56], or Araneola [57], are examples of protocols that can implement this interface to make available to applications (and other protocols) a dissemination service.

In this work we consider the existence of a Disseminate operation that should be implemented by protocols providing the service.

***Operations***

**Disseminate(byte[] data)** The Disseminate operation allows to delegate the dissemination of data to some dissemination decentralised service by an application (or other protocol). The guarantees of this communication primitive are protocol dependent. The operation considers only one parameter, data, which contains the information to be disseminated. As we do not make any assumption on the type of information that can be disseminated by protocols providing this service, the data parameter can be considered as an array of bytes.

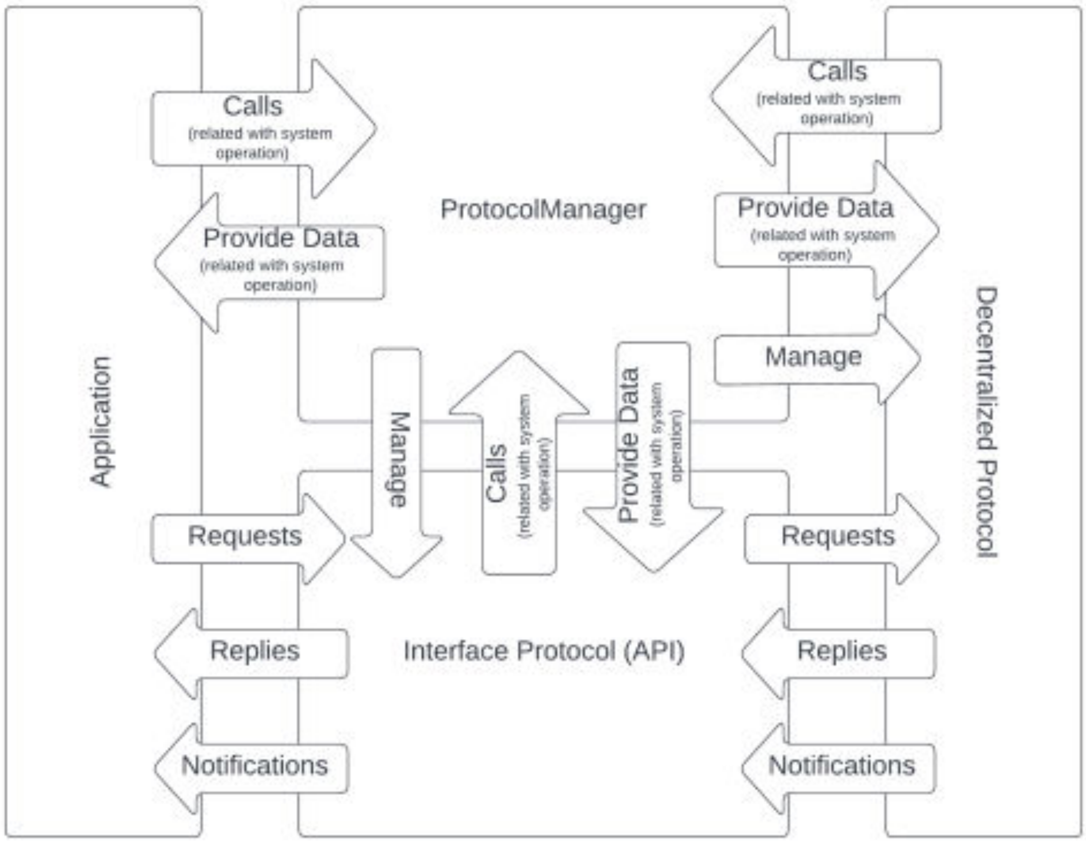
***Notifications***

**DataReceived(byte[] data)** A DataReceived notification can be triggered by protocols, to applications/other protocols that consume information received from the Dissemination service. The notification is triggered whenever  new data, disseminated by any node on the network, is received. The DataReceived notification contains only the data field, a byte array responsible for storing the disseminated data.

#### 2.2.1.3 Architecture and Implementation

In this section we briefly discuss the proposed architecture and components to provide generic decentralised services that can be instantiated only at runtime, removing the dependence of applications from specific protocols. Our solution consists of multiple components, working together to provide an abstraction layer between applications and decentralised protocols. The purpose of this abstraction layer is to provide a generic, standard, and simple way for allowing applications to interact with decentralised protocols, where only the generic interfaces (presented above) are exposed.

Additionally, the solution presented here also aims at simplifying the choice and instantiation of the most adequate protocol, providing the decentralised services required by an application, as well as allowing a simple management of the multiple decentralised protocols being executed simultaneously to provide the required services within a single process.


Application
Calls (related with system operation)
Provide Data (related with system operation)
ProtocolManager
Calls (related with system operation)
Provide Data (related with system operation)
Manage
Manage
Calls (related with system operation)
Provide Data (related with system operation)
Decentralized Protocol
Requests
Requests
Replies
Interface Protocol (API)
Replies
Notifications
Notifications

Our solution consists of four main components, illustrated in the figure above, that also captures the main interactions between these components. The main components of our solutions are the following (which we present in more detail in the following).

- The Protocol Manager, which is the core of our solutions that bridges and mediates all other components;
- A set of generic interfaces for allowing applications to interact with decentralised protocols;
- The decentralised protocols providing a set of services, which is an extensible set of protocol implementations;
- The applications that require services provided by the protocols, where for generality we assume that multiple applications can be executed within the context of the same process.

#### *2.2.1.3.1 Protocol Manager*

The Protocol Manager component is responsible for managing all the other components. In our solution we consider that for each process only one instance of the Protocol Manager exists. All other components interact with the local Protocol Manager to request and access resources or information about the system. We have implemented a simple prototype and demonstrator of this approach, as an extension to the Babel framework, hence the listings below are in Java (the programming language used to develop Babel) but the described approach can be implemented in any programming language.

Among other relevant information, the Protocol Manager maintains information about:

- The available decentralised protocols, as well as the services provided by each one;
- The available services (and interfaces) for interacting with decentralised protocols;
- The decentralised protocols running at a given moment;
- The generic interfaces instantiated, at a given moment, to enable the interaction between the applications and each executing protocol;
- Global configuration parameters and properties requested by application.

The Protocol Manager provides applications (and other protocols) the means for instantiation of a new decentralised protocol. This can be achieved by providing the protocol name (e.g., Kademlia) or the service interface required by the application/protocol (e.g., Routing). The interfaces exposed by the Protocol Manager to allow protocol instantiation are presented in the two listings below.

```
<Integer,Set<GenericAPI>> newProtocol
   (DecentralizedProtocol protocol, Integer protocolId,
   Boolean useExisting, Host host,
   Properties runProperties)
```

```
<Integer,Set<GenericAPI>> newProtocol
   (Set<Services> servicesSet,
   Integer protocolId, Boolean useExisting, Host host,
   Properties requiredProperties, Properties runProperties)
```

The main difference between these two API is that on the first one, the programmer indicated a specific protocol to be instantiated (parameter protocol), while on the second the programmer provides a set of Services that the protocol should provide. The latter is the more interesting one, since it fully decouples the logic of the code being written of any specific protocol.

We note that, both methods require the requester to provide an integer named protocolId, Applications, as well as other system components, can rely on this identifier to refer to that specific instance of the protocol, which for instance allows for different instances of the same protocol to co-exist at runtime, using different parameters, and used for different purposes.

These methods return the protocolId and a set of service APIs, since a single protocol, as discussed previously, can implement more than one decentralised service. The requester can then use these APIs to interact with the multiple services provided by the protocol.

In addition to the parameters discussed above, these APIs all allow present the following parameters:

**useExisting** The useExisting parameter can be used by applications to indicate what should happen if an instance of the requested protocol is already running. If this parameter is set to true and an instance of the requested protocol is already running, the Protocol Manager does not create a new instance of the protocol and, instead, returns information about the existing one. If this parameter is set to true, applications should check the identifier of the protocol returned (protocolId) as a result of the newProtocol operation, because it might not be the one requested if another one is already running.

**host** The host parameter allows applications to specify the host information, e.g., IP address and port, that the protocol should use for its operation. If the useExisting parameter is set to true and a protocol instance is already running, this parameter is ignored.

**runProperties** The runProperties parameter allows to specify the static configurations that a protocol should use when running.

**requiredProperties** The requiredProperties parameter enables applications to provide a set of properties to be used when choosing the protocol to be instantiated, between all protocols that provide the services requested in the servicesSet parameter. The properties provided in the requiredProperties parameter differ from the ones in runProperties: whereas the first ones are related with each decentralised service and allow choosing the most appropriate protocol for a specific scenario, the second one is related with the protocol execution, as explained before.

##### *2.2.1.3.2 Generic Interfaces and Different Programming Models*

The generic interfaces are special protocols (using the Babel terminology) that mediate the interaction between applications and specific decentralised protocols. Each interface consists of a set of operations, notifications, and properties, related with a specific decentralised service. While we can consider several services, in our prototype we consider the service interfaces presented previously (see Section 2.2.1.2.1).

When a protocol is instantiated (by any of the mechanisms discussed above), all service interfaces associated with the protocol are also created (remember that a single decentralised protocol can provide multiple decentralised services, for instance Chord support *Application-level Routing*, *Membership Management*, and *Resource Storage*).

If a different application or protocol needs to access the service exposed by a protocol already in execution, it must request the instance of that decentralised service using the protocol identifier from the Protocol Manager. All interactions with that concrete protocol will be made through the appropriate service interface.

While Babel, and the strategy currently put forward by the TaRDIS project and discussed in Deliverable D3.1 [58], is to take advantage of a programming model based on state machines, where the interaction between components happen through asynchronous events, this approach is sometimes cumbersome to programmers, particularly when two components interact using a request-reply model, where component A requests something of component B that it requires to complete the task it is currently executing.

Consider the following concrete example. A gossip-based protocol - say *protoA* - receives a request from an application to disseminate a message *m.* However, to do this, *protoA* will have to interact with a Membership Management service (another protocol) - say *protoB* - to get a sample of other participants in the system to whom it can forward message *m*. To achieve this, since the interaction between *protoA* and *protoB* is achieved exclusively through asynchronous events, it will force *protoA* to:

1. Upon receiving the application request to disseminate *m*, *protoA* must store *m*, generate a requestID which is also stored associated with message *m*.
2. Issue a GetNeighborsSample request to the Membership Management service API that will be asynchronously delivered to *protoB*.
3. When *protoA* receives the GetNeighborsSample reply from *protoB* at some point in the future it will have to extract the requestID in that reply, match that requestID to the message *m* that he previously stored, and forward that message to the elements in the sample also contained within the GetNeighborsSample reply. Only after this is the handling of the application request completed.

While this interaction model allows protocols to handle other requests while additional information is gathered from other components and allows to create a complete isolation between the state managed by each individual protocol, this is from a programming point of view, time consuming, verbose, and error prone. To overcome this limitation of this programming model, we have decided to experiment with supporting three distinct interaction models between components (in this case protocols) using a single implementation and dealing with this at the Generic Service Interface. Therefore we support the following three interaction patterns:

- A fully asynchronous interaction based on event-driven mechanisms with requests and replies;
- A Promise-based interaction where requesters receive a promise (or future) and can execute other tasks until they block until the promise can be resolved (i.e., they can block their execution until the reply is received;
- A synchronous (blocking) interaction where the requester always waits until the operation is complete whenever they execute a request to another component.

Considering a wide range of interaction mechanisms provides programmers with different possibilities of interaction. This also shows that it is possible to have adaptors that transparently expose a different interaction model, while the asynchronous event-based model is used to implement and govern the execution of different protocols.

Under the fully asynchronous interaction model, protocols and applications are implemented by creating and registering handlers that are automatically triggered whenever any event happens. These events can be a request received from some other component, or a reply received form a protocol in response to a request that had been previously triggered, or an asynchronous notification that something relevant has happened[4].

The promise-based interaction model allows implementations to take advantage of the constructions present in many programming languages, such as Java [21], Scala [24], C++ [20], or Rust [22], where an asynchronous operation can return a Promise (or Future) that can later be tested (or blocked on) for completion. This allows applications to control the amount of concurrency and defer the execution of operations, e.g., by requesting an operation, obtaining a Promise, performing some additional computations, and verifying later if the operation is completed and the results are available to continue its execution.

In the synchronous interaction model, applications will immediately block and wait for the completion of a request operation that generates a reply. This mechanism, although not allowing any type of concurrent execution, provides developers with a simpler and less error-prone model of interaction as they do not need to define handlers for the results of asynchronous operations nor deal with Promises.

Even though, as stated before, three types of application-interface interaction models are possible within our solution, some operations, due to its nature, might not provide all interaction options. As an example, we can consider the Disseminate operation of the Dissemination service interface. This operation does not return any value to the application requesting it, as it triggers the dissemination of data throughout the system by delegation of that task to a protocol. In this case, in general, only one option is available for requesting the operation, as there is no distinction between synchronous and asynchronous implementations.

[4] There is also an additional type of event that we consider in Babel [59] named timers. Timers allow a protocol to execute some behaviour (i.e., execute an event handler) when some timeout or periodic timer occurs.

#### *2.2.1.3.3 Decentralised Protocols*

The decentralised protocols are the providers of functionality (i.e., the components that effectively materialise decentralised services). As an example, if we consider the Kademlia or Chord protocols they can provide the operations related with the Application-level Routing service (among others). Considering our implementation in Babel, existing decentralised protocol implementations had to be modified to make them compatible with our approach, we now briefly explain what are the main requirements for protocol implementation to use our approach.

Protocols need to implement the logic to handle requests, process them, and return results, trigger the notifications exposed by the service interface, and handle any other type of events (potentially generated by other protocols) that are essential to ensure their correct operation. Although not mandatory, properties related with each implemented service also need to be defined and exposed by the protocol logic, to guide applications in choosing the most suitable protocol for their operation.

Some additional restrictions are also applicable to the implementation of methods and parameters used when instantiating the protocols. These restrictions depend on the implementation of the Protocol Manager and are related with how the necessary parameters are transmitted to the protocol, e.g., the protocol identifier, the host information, or the configurations related with the protocol operation. In fact, although this might be implementation-dependent, we expect the way each protocol is instantiated, i.e., the mechanisms used on instantiation and the initialization, to be the same across all protocols instantiated by a process of a given swarm.

As with all other components in the system, protocols can also rely on the Protocol Manager to obtain run-time information about the system operation, namely information about other running protocols, system configurations, access to generic interfaces to interact with other protocols, among others.

#### *2.2.1.3.4 Applications*

The applications are the system components that primarily[5] act as a client for a set of decentralised services, provided by decentralised protocols. Applications obtain from the Protocol Manager (references for) instances of protocols providing the services required for their operation, both by requesting a new instance to be created, or by obtaining references for a previously instantiated protocol. Applications, as well as any other component, can also request from the Protocol Manager relevant run-time information about the system operation.

As explained before, applications interact with the decentralised protocols by issuing events that are of the type Request and registering handlers, through the generic interfaces provided, that handle events generated as responses to those requests, or asynchronous notifications. Consequently, application programmers can change the protocol providing a given service to another one, just by modifying the instantiation call to the Protocol Manager, and without making other changes on their code, as all interactions with protocols are executed through generic (and protocol-independent) interfaces. This requires a discipline on developing application code, where applications never interact directly with protocols and, as such, requests for obtaining a protocol instance or to execute an operation must always be performed, respectively, through the Protocol Manager or the corresponding generic service interfaces.

[5] We remind the reader that a decentralised protocol itself might use a decentralised service exposed by another decentralised protocol, akin to how protocols in the OSI network stack can use the functionalities provided by other protocols in lower layers of the OSI stack.

#### 2.2.1.4 Experimental Validation

The architecture and solution described here was implemented as a Java prototype by modifying the Babel framework - we discuss this framework in Section 3.2 - by creating a fork of Babel which we provide. We further discuss the implementation of this solution in Section 1 of this report. We note that the results reported here have guided the evolution of tools for the main version as Babel, that we also discuss in Section 1.

We now present the performance evaluation of our approach by comparing both routing and dissemination-based applications against their respective versions that only rely on the Babel framework [59]. To evaluate the performance of our solution each application was executed for a predefined amount of time and a set of relevant performance metrics, described in detail in each of the following sections, were measured. The results presented for each of the test conditions were, in all cases, obtained from an average of multiple test executions and, when deemed necessary, the confidence intervals (with a level of confidence of 95%) are also presented. Also, the retrieval of the metrics for each test was performed by parsing the logs obtained from each execution, after the test has been completed, to not contaminate the execution of the tests with the load of computing or storing metric-related data.

A comparison between initialization times was also performed to assess the impact of the Protocol Manager and the Generic Services programming interfaces. We consider the initialization time as the period between the request for the instantiation of a protocol, both through the Protocol Manager or directly by interacting with the Babel core, and the moment when the protocol is available to be used by an application.

##### *2.2.1.4.1 Routing Application*

We have developed a simple routing application that is responsible for performing routing requests of randomly generated queries through a swarm system, by relying on a routing protocol like Kademlia. The application expects to receive the set of nodes matching the query provided. In our evaluation the Kademlia protocol was executed with the following configurations: $k\ value$ = 20, $alpha$ = 3, and the timeout for node lookups was configured with a value of 10 seconds. The routing requests were carried out by the application, in a closed-loop, for 4 minutes with a start and cooldown period of 2 minutes each (hence each experience had a total duration of eight minutes).

***Evaluation Metrics***

When evaluating the performance of the routing application, the following metrics were considered.

**Number of requests sent** The number of requests sent by the application for performing routing operations, each with a different randomly generated query.

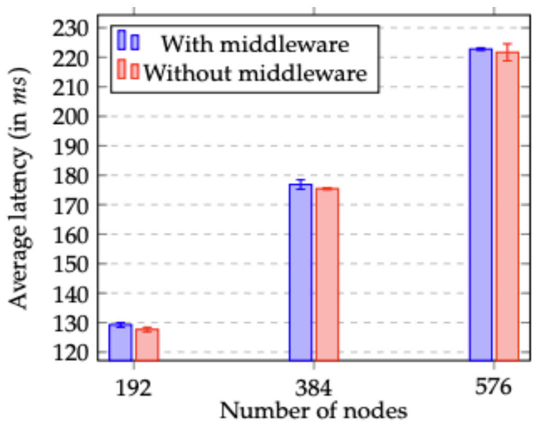
**Number of results received** The number of results received, from the routing protocol, containing the set of nodes that result from executing the provided query.

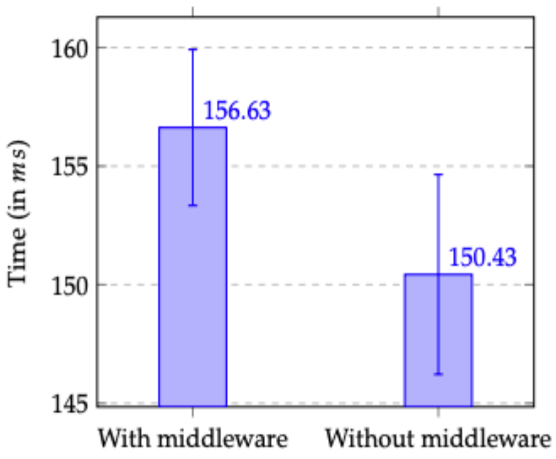
**Average Latency** The average latency, in milliseconds, for receiving the routing results in relation to the time at which the request was performed.

**Throughput** The throughput is presented in responses per second and represents the rate of routing responses arriving in a given time interval (1 second in this case). The throughput was obtained by dividing the number of results received by the period of testing (in seconds).

**Recall** The recall represents the fraction of values that are considered correct across all results that were returned by the routing protocol. This metric was calculated by verifying, for each result obtained, if the returned set of nodes was correct and, then, dividing the number of correct results by the total number received results.

***Results***





The experiments with the routing application were performed by running independent Java processes in a computational cluster. We executed experiments with 192, 384, and 576 processes split, respectively, by 3, 6 and 9 machines running 64 nodes each. Each test was executed three times and presented results are averaged across executions. The comparison of latencies between the versions of the routing application, with and without relying on the developed abstractions is presented on the figure above at left, whereas the figure at right provides a comparison between the initialization time of both application versions. The initialization time was obtained by calculating an average of the initialization times considering 192 nodes split into three machines. The results concerning the remaining metrics collected in these experiments are provided in the table below.

| Application | Nodes | Requests Sent | Responses Received | Latency ($ms$) | Throughput ($responses/s$) | Recall |
|---|---|---|---|---|---|---|
| With middleware | 192 | 353151.3 | 353151.3 | $129.267 \pm 0.834$ | 1128.3 | 1.0 |
| | 384 | 517837.7 | 517837.7 | $176.848 \pm 1.650$ | 1580.4 | 1.0 |
| | 576 | 617627.3 | 617627.3 | $222.793 \pm 0.370$ | 1800.7 | 1.0 |
| Without middleware | 192 | 357782.3 | 357782.3 | $127.686 \pm 0.723$ | 1143.1 | 1.0 |
| | 384 | 522143.3 | 522143.3 | $175.432 \pm 0.334$ | 1591.9 | 1.0 |
| | 576 | 620851 | 620851 | $221.686 \pm 2.901$ | 1810.1 | 1.0 |

Through the analysis of the results presented in the table and the rightmost figure it is possible to conclude that, although the use of our solution incurs in a slight increase on latency and, therefore, a reduction in throughput, the overall impact is quite small. In fact, when considering the confidence interval, both solutions might even be considered equivalent performance-wise. This result is not surprising, since the impact of the network on these performance metrics is expected to be orders of magnitude above the overhead introduced by our solution. We note that the recall value of 1.0, obtained on all experiments confirms that both the implemented decentralised protocols and all other components of our solution are exhibiting a correct behaviour, hence, as expected, our approach has no effect on the correctness of protocols.

Regarding the time required for protocol initialization, the results show that the time to initialise a protocol using the proposed abstraction is, not surprisingly, slightly above than the one measured when using the unmodified Babel framework. This is expected since the mechanisms for protocol instantiation are more complex in our solution, however, as shown before this impact on the overall performance becomes negligible since instantiation of protocols happens, most of the time, only once when a new participant is joining the swarm system.

##### *2.2.1.4.2 Dissemination Application*

We also have developed a simple dissemination application to further evaluate our solution. This application operates by having each process in the swarm system to periodically request the dissemination of a message with a configurable size. The application generates an identifier for each message (an UUID) and logs the moment when the dissemination request was performed. Then, when a message is received, the identifier is obtained and logged together with a timestamp.

For this application we leveraged on Plumtree as dissemination protocol and on HyParView as the underlying peer sampling protocol. The Plumtree timeout, for considering a connection as failed, was configured to 7 seconds, while the timeout for awaiting a GRAFT response was defined to 3.5 seconds. The HyParView protocol relied on active and passive views of size 5 and 12, respectively. The application performs the dissemination of a message every 30 seconds, with a size of 100 KB, during a period of 4 minutes with a start and cooldown period of 2 minutes each (similarly to the previously reported experiments, each of these experiments lasted for a total of eight minutes).

***Evaluation Metrics***

When evaluating the performance of the dissemination application, the metrics described below were considered:

**Number of messages sent** The total number of messages requested by the application to be disseminated throughout the network, relying on the underlying protocols. For each request, a payload with the pre-configured size was disseminated.

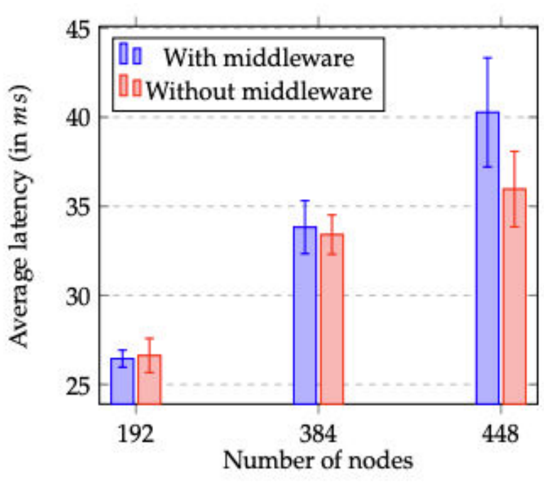
**Total delivered messages** The total number of messages delivered on the various instances of the dissemination application (across different processes). As an example, if a message is disseminated by a node on a network containing 8 nodes, and all of them receive the message, we consider the total delivered messages is 8.

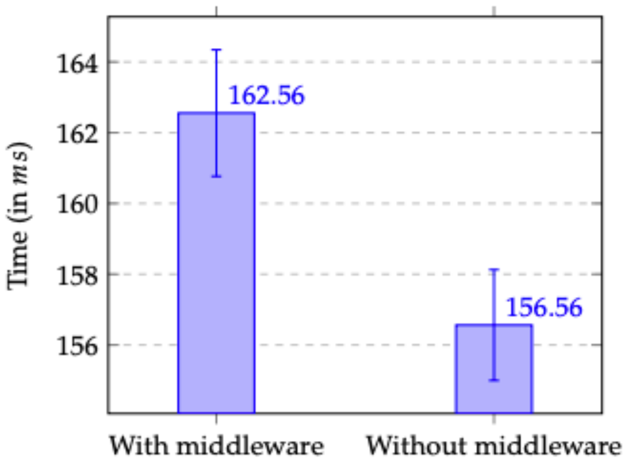
**Average Latency** The average latency was considered as the average of the maximum latencies obtained for each disseminated message. To retrieve this metric the maximum latency for each message was obtained by calculating the difference between the moment when the message was last delivered and the moment when the dissemination request was made. Then, an average of the maximum latencies was performed to obtain the final result.

**Throughput** The throughput is presented in messages per second and represents the rate of messages delivered on the system in a given time interval (1 second in this case), averaged across all nodes in the system.

**Reliability** The reliability represents the fraction of disseminated messages that are correctly delivered to nodes present in the swarm system. In our evaluation we calculated the reliability for each message by dividing the number of nodes on which the message was delivered by the total number of network nodes. An average of the reliability values was then performed to obtain the result.

***Results***





The evaluation of the dissemination application was performed by executing 192, 384, and 448 processes distributed across 3, 6, and 7 machines respectively, in a computational cluster. Each test was repeated ten times and the results presented here, for each set of test parameters, are the average of the results obtained across these independent executions. The higher number of tests performed, in comparison with the routing application, is explained by the higher variation in the obtained results. The figure at the left above shows the comparison between the latency values of both implementations of the dissemination application, with and without relying on our solution, whereas the figure to the right presents the comparison between the initialization time of protocols on both implemented versions (considering 192 processes split across three machines). Finally the table below summarises the results obtained for the remainder measure performance indicators.

| Application | Nodes | Messages Sent | Total Delivered Messages | Latency ($ms$) | Throughput ($messages/s$) | Reliability |
|---|---|---|---|---|---|---|
| With middleware | 192 | 1536 | 294912 | 26.445 ± 0.488 | 614.4 | 1.0 |
| | 384 | 3072 | 1179648 | 33.821 ± 1.497 | 2457.6 | 1.0 |
| | 448 | 3584 | 1605632 | 40.273 ± 3.052 | 3345.1 | 1.0 |
| Without middleware | 192 | 1618.7 | 310732.8 | 26.624 ± 0.946 | 647.4 | 1.0 |
| | 384 | 3221.7 | 1237132.8 | 33.410 ± 1.088 | 2577.4 | 1.0 |
| | 448 | 3757.6 | 1683404.8 | 35.962 ± 2.107 | 3507.1 | 1.0 |

Considering the results in the table and figure to the left it is possible to observe similar results to the ones previously discussed for the routing application. There is no significant difference in the performance of both implementation alternatives, although as expected the additional mechanisms of our solution to enforce the generality of interfaces exposed by concrete decentralised protocols do introduce a small overhead that can be can be observed on the results, and that grow slightly with the size of the system, which makes sense, since messages will have to be forwarded a larger number of times for larger systems, which increases the effect of the overhead of our architecture.

When considering the results regarding the initialization time, presented in the right-most figure, we can observe similar results to the ones discussed for the routing application. By comparing the initialization times of both implementations of the dissemination application it is possible to conclude that the initialization of a new protocol on the implementation relying on our solution takes slightly longer than the native Babel counterpart.

Funded by the European Union

#### 2.2.1.4.3 *Code complexity evaluation*

| Applications | Lines of Code | | % of reduction |
|---|---|---|---|
| | With middleware | Without middleware | |
| Peer Sampling | 29 | 64 | 54.7 % |
| Dissemination | 70 | 132 | 46.9 % |
| Routing | 66 | 120 | 45 % |
| Resource Storage | 202 | 360 | 43.9 % |

For comparing the code complexity between the application versions with and without relying on the programming interfaces and management mechanisms proposed here, we performed an analysis over the number of lines of code required for implementing each version of the applications. Here we consider two additional applications that we have developed, a first one that is based on providing sample of nodes in the swarm (Peer Sampling) and another, which is more complex, that allows nodes to store, access and edit files that are stored on a DHT, which notifies nodes that accessed the file whenever it is updated by using a dissemination service (Resource Storage).

When counting the lines of code required to implement each version of the applications the blank lines and comments were not considered as well as the lines related with imports and package definitions in Java classes. In the comparison between both versions of the resource storage application, the lines related with the user interaction for performing the operations were also not considered.

The results regarding the code complexity of each application are presented on the table above, which shows that overall the number of lines of code required to implement each application using our approach is indeed smaller than when using the native Babel (the table indicated the percentage of lines of code reduction). This improvement is explained by the simple mechanisms for protocol instantiation as well as the common programming interfaces exposed, providing multiple synchronous and asynchronous interaction mechanisms between protocols and applications.

As an example, when considering the Routing application, the version relying on the abstractions developed can launch a thread, active on the period during which the routing requests should be performed, to issue FindNodes operations in a closed-loop. The code leverages on the Futures-based interaction mechanism, requesting the operation and blocking on the returned future until the return value is available or a timeout expires. The simple instantiation of a protocol just by calling an operation exposed by the Protocol Manager, therefore obtaining the programming interfaces to interact with it and perform the necessary operations is also an advantage. Additionally, being able to implement the application without having to interact with the Babel mechanisms simplifies the development both by reducing the learning curve for the programmer and the complexity of code, as no initialise methods, specific constructors, nor event handlers are required because the application does not need to be implemented as a Babel protocol.

Conversely, the version of the application that does not rely on the abstraction layer, due to the necessity of development as a Babel protocol, needs to implement all logic based on the asynchronous mechanisms, exposed by Babel, through the implementation of handlers responsible for dealing with the replies from the FindNodes operation. The application is also required to register those handlers in Babel. To implement the same closed-loop behaviour, the handler should send the next FindNodes request to the protocol when the last one is received. Moreover, the start and stop of the routing requests needs to be managed by a Babel timer, which requires the development of one more Java class and the respective timer handler.

When considering more complex examples, like the Resource Storage application, the advantages of our solution become even more evident as not only the number of lines of code is reduced, but also the implementation of the application becomes significantly easier for the developer. As an example, when a node wants to add a new file a set of operations might need to be performed that include instantiating a protocol to disseminate the notifications related to the file, performing a Join operation in the dissemination protocol using the nearest nodes to the file identifier as contact nodes, and disseminating the notification throughout the network.

Implementing the logic described before requires coordination between distinct protocols and operations. An example of coordination is obtaining of the nearest nodes to the file identifier, which needs to be done before requesting the Join operation from the dissemination protocol, as the nodes retrieved will be used as contacts. The coordination between different operations leverages on the blocking operations exposed by our solution instead of dealing with an asynchronous interaction model that requires more complex logic and is more error-prone.

Additionally, in more complex applications that require multiple decentralised protocols to work properly, having a single component for managing all of them (in our proposed solution the Protocol Manager) is also an advantage. Moreover, relying on generic interfaces for interacting with the services provided by the protocols not only simplifies the development, but also contributes to the maintainability by allowing the change from one protocol to another without profound application changes.

#### 2.2.1.5 Summary

In this activity we performed a study over a set of decentralised protocols, that strived to identify decentralised services provided by them. For each of these services we identified which operations and properties could fully capture their operation in a protocol-independent way. Based on this we devised a set of generic abstractions (or programming interfaces) that can be leveraged to interact with decentralised protocols using a service-based approach instead of a protocol-based one. Our solution provides multiple interaction mechanisms for requesting operations from decentralised protocols, employing both synchronous and asynchronous approaches. Programmers can also request a decentralised service to be provided even without knowledge of the specific protocols providing it, as the instantiation of protocols can be requested just by defining the service(s) and (optionally) the properties required. In summary, by combining the devised interfaces with the mechanisms developed for managing decentralised protocols running on a system, a middleware solution based on multiple components was developed.

We provided a reference implementation of the solution proposed, developed in Java, based on the Babel framework. Finally, an evaluation was performed, leveraging on the implementation mentioned before, to assess the impact of the solution on the performance and code complexity of applications. The results showed that the improvements in terms of code complexity are noticeable without a relevant impact on key performance indicators of applications.

### 2.2.2 An Epidemic and Scalable Global Membership Service

While scalability and availability are primary concerns in swarm systems, and this is usually achieved by avoiding a central point of control and taking advantage of decentralised and scalable membership services [19,22,25,26], in some scenarios, non-essential tasks could benefit from having an eventually correct notion of a global membership. This can be useful for instance to collect statistics from the operation of the system, or to feed information for consoles that report the status of the system to human operators. Unfortunately, we have not found in the literature a decentralised membership service that could provide an eventually correct view of the system membership to nodes in the system, Eventually correct in this context means that the global view reported by the system will be correct after a long enough period with no changes to the membership and where the network behaves in a synchronous way (i.e., there are no network partitions, and nodes can effectively communicate within an arbitrary large configurable time window).

To overcome this limitation, and because this functionality has been identified as being of relevance for industrial use cases of smart factories [96] we developed a new solution that has interesting features. In particular, in addition to providing a global (i.e., complete) eventually correct membership view to each node, our protocol takes advantage of a partial view based membership abstraction, and a decentralised broadcast communication abstraction. We do not prescribe what should be the concrete membership and communication protocol that should be employed for the operation of this novel membership service, being sufficient that the membership service ensure global connectivity at the random graph denoted by the overlay maintained by that service, and the broadcast protocol provides probabilistic atomic broadcast with a configurable high probability [22], meaning that with a high probability all broadcast messages are eventually delivered by all correct processes. We named this new protocol *Epidemic Global View*.

**Algorithm 1:** Epidemic Global View (Interface and State)

```
1:  Interface:
2:     Requests:
3:        GetNeighborsSampleRequest ( $ss$) //$ss$ maximum size of the sample
4:     Replies:
5:        GetNeighborsSampleReply ( $sample$, $ss$)
6:     Notifications:
7:        NeighborUp ( $p$ ) //Notification that a new peer $p$ was added to membership
8:        NeighborDown ( $p$, $suspect$ ) //Notification that a peer $p$ is no longer in the
9:           membership, $suspect$ is a boolean flag that indicates a suspicion that
10:          the $p$ has failed

State:
11:    self identifier of the local node with a logical time stamp
12:    membership //set with all neighbors identifiers
13:    tombstones //set with all past neighbors identifiers

14:    Upon Init ($myself$, $T_{ka}$) do:
15:       self ⟵ (myself , 0)
16:       membership ⟵ { self }
17:       tombstones ⟵ { }
18:       Setup Periodic Timer keepAliveTimer ( $T_{ka}$ )
```

### 2.2.2.1 Protocol State and Initialization

Algorithm 1 (above) shows the proposed interface, internal state, and the special init event handler (we assume that when an instance of this protocol is started in some process, the init event is automatically triggered, providing the protocol with runtime configuration parameters). The protocol exposes the simplified membership protocol interface that we derived from the work presented previously on Section 2.b.i. We have simplified the proposed interface to make it more convenient and less intrusive for developers, and we materialised this interface in the Babel framework [59] (which we discuss further ahead on Section 3.b). The API has a single request, which is to request a sample of elements in the current membership with a size of up to *ss* (Alg. 1, line 3) which has a corresponding reply that can be generated by the protocol (Alg. 1, line 5). Additionally, the protocol can emit notifications NeihgborUp and NeighborDown (Alg. 1, lines 7-8) which respectively indicated that the protocol has become aware of a new peer in the system or suspects that a previously known peer has failed (in this protocol, the flag suspect of the NeighborDown notification is always true), respectively.

The protocol state is quite simple, and the design of the protocol is inspired by the design of some conflict-free replicated data types (CRDTs) [] where elements that were previously on the set are kept as metainformation to avoid their incorrect radiation to the set by a delayed past operation. In our protocol this translates to the following design. Each node identifier (we recall that typically these identifiers provide the information necessary to contact some process in a network, such as IP, port, transport protocols that can be used to contact the node, and potentially a logical identifier) is enriched with a numerical timestamp, that acts like a logical timestamp, initially this timestamp is set to zero (Alg. 1 line 15), and the timestamp for a particular peer p can only be incremented by *p* (i.e., when a process identifier is forwarded among peers, they maintain the timestamp associated with that identifier).

The (current) local view of a process of the system membership is kept on the membership set that contains process identifiers (enriched with the logical timestamp) including the one for the local node (Alg.1 line 16). Processes that have been previously known (i.e., at some point their identifier was in the membership set) but that meanwhile have been suspected of failing (we detail how this happens ahead) are moved to the tombstone set. Notice that when moving a process identifier from the membership set to the tombstone the timestamp associated with that process is kept unchanged. A process identifier (independently of the timestamp value associated with it) can only be in one of the membership or tombstone sets of a process (which means that at any given point in time a process will, for processes that they have been aware at some point, either believe it to be correct or faulty. The last instruction in the Init event handler of Algorithm 1 (line 18) is to setup a periodic (local) event that allows the protocol to do actions, with the period of this action being a parameter of the protocol.

#### 2.2.2.2 Protocol Operation

Algorithm 2 (below) shows the event handlers of the other events associated with the operation of the Epidemic Global View protocol. We start by the actions that are taken periodically by every process executing this process which is captured by the event handler of the keepAliveTimer (Alg. 2, lines 7 – 11). This periodic action has two complementary objectives, the first is to increment the timestamp associated with the identifier of the local process and broadcast it throughout the network in a ALIVE message. This serves the purpose of making the local process known to other processes that have joined the system recently (i.e., after the last broadcast of the local process), and to prove to other processes in the system that the local process remains active. The other purpose is to suspect processes that have not disseminated an ALIVE message for a long enough period. For simplicity we assume this to be 3 times the value keepAliveTimer period, although this could be adjusted, for instance to allow different processes to broadcast their ALIVE message less frequently, by issuing with this message the period over which that identifier should be considered valid after reception. If a process suspects a process of being faulty due to lack of observable activity by that process it broadcasts a SUSPECT message throughout the system containing the identifier of the suspected process (which we remind the reader contains the last observed timestamp for that process).

The core of the operation of the protocol is related with the way that processes handle the reception of ALIVE and SUSPECT messages respectively. Upon receiving an ALIVE message (Alg. 2 lines 16 – 26) a process will execute the following steps. If process being announced if the ALIVE message is already part of the local process membership process and the timestamp of the identifier in the membership set is lower than the timestamp of the identifier within the ALIVE message, the process updates its local membership set to reflect the higher timestamp value (and it also records the time of reception of this message, this was omitted from the pseudo-code for readability). If the process identifier was contained within the tombstones set (i.e., that process had been suspected) with a timestamp value below the timestamp value of the identifier within the ALIVE message, then the local process removes the identifier of the process that issued the ALIVE message from the tombstones set and adds the received identifier to the membership set. This implicitly means that the process is no longer suspected of being faulty. Finally, the process that broadcasted the ALIVE message was not in the local membership nor the tombstones sets, this means that this process was unknown until this point, and the received identifier is added to the membership set.

**Algorithm 2:** Epidemic Global View (Main)

```
1:  Upon GetNeighborsSampleRequest ($ss$) do:
2:    If # (membership \ self) <= $ss$ Then
3:      sample $\longleftarrow$ membership \ self
4:    Else
5:      sample $\longleftarrow$ pickRandom( ( membership \ self) , $ss$ )
6:      Trigger GetNeighborsSampleReply( sample, #sample)

7:  Upon Timer keepAliveTimer do
8:    self $\longleftarrow$ incrementTimestamp(self)
9:    Trigger BroadcastRequest (ALIVE, self)
10:   For $p \in$ membership: lastUpdateTime($p$) + 3 $\times T_{ka}$ do
11:     Trigger BroadcastRequest (SUSPECT, $p$)

12: Upon NeighborDown($p$) do
13:   If $\exists v \in$ membership: id($v$) = id($p$) Then
14:     trigger NeighborDown(id($p$), true)
15:     Trigger BroadcastRequest (SUSPECT, $v$)

16: Upon Receive (ALIVE, $p$) do
17:   If $\exists v \in$ membership: id($v$) = id($p$) Then
18:     If timestamp($p$) > timestamp($v$) Then
19:       membership $\longleftarrow$ membership $\setminus \{v\} \cup \{p\}$
20:   Else If $\exists v \in$ tombstones: id($v$) = id($p$) $\wedge$ timestamp($p$) > timestamp($v$) Then
21:     tombstones $\longleftarrow$ tombstones $\cup \{v\}$
22:     membership $\longleftarrow$ membership $\cup \{p\}$
23:     trigger NeighborUp(id($p$))
24:   Else If $\nexists v \in$ tombstones: id($v$) = id($p$) Then
25:     membership $\longleftarrow$ membership $\cup \{p\}$
26:     trigger NeighborUp(id($p$))

27: Upon Receive (SUSPECT, $p$) do
28:   If id($p$) = id(self) Then
29:     membership $\longleftarrow$ membership $\setminus \{$self$\}$
30:     While timestamp($p$) $\leq$ timestamp(self) do
31:       self $\longleftarrow$ incrementTimestamp(self)
32:     Trigger BroadcastRequest (ALIVE, self)
33:     membership $\longleftarrow$ membership $\cup \{$self$\}$
34:   If $\exists v \in$ membership: id($v$) = id($p$) Then
35:     If timestamp($p$) $\geq$ timestamp($v$) Then
36:       membershp $\longleftarrow$ membership $\setminus \{v\}$
37:       tombstones $\longleftarrow$ tombstones $\cup \{p\}$
38:       trigger NeighborDown(id($p$), true)
39:   Else If $\exists v \in$ tombstones: id($v$) = id($p$) $\wedge$ timestamp($p$) > timestamp($v$) Then
40:     tombstones $\longleftarrow$ tombstones $\setminus \{v\} \cup \{p\}$
41:   Else If $\nexists v \in$ membership: id($v$) = id($p$) Then
42:     tombstones $\longleftarrow$ tombstones $\cup \{p\}$
```

The reception of a SUSPECT message (Alg. 2, lines 27 – 42) has a somewhat symmetric behaviour to that of the ALIVE message, where if the process being suspected, was known, and already suspected, and the received message contains an identifier with a higher timestamp value we update the tombstones set to reflect that (Alg. 2, lines 39 – 40), and if the node whose identifiers is being carried on the SUSPECT message was previously in the membership set, with a timestamp lower or equal to the one of the received identifier, that node is moved from the membership to the tombstones set and becomes locally suspected

(Alg. 2, lines 34 – 38). There is another case which is a node that was previously unknown, which is simply added to the tombstones set, but without triggering the local NeighborDown notification (Alg. 2, lines 41 – 42).

More importantly, and a key difference in relation to the ALIVE notification, is that if a process receives a SUSPECT notification for itself, it immediately takes corrective measures (Alg. 2, lines 28 - 33). In particular, if this happens the process will start by increasing the timestamp associated with its own identifier until it surpasses the timestamp in the identifier carried by the SUSPECT message. After this it immediately propagates an ALIVE message for himself with its identifier updated, to ensure that any process in the system that delivered (or will deliver) the incorrect SUSPECT message keep the local process as part of their membership sets, and consequently as part of the system.

#### 2.2.2.3 Future Work

We note that the algorithm presented here - and that we implemented in Babel as reported further ahead on Section 3.8 - will be evolved in the future. As discussed above, the mechanism to suspect that a process is faulty is based on a local perception of time, but this requires every process to propagate ALIVE messages using the same period of time, which might not be beneficial in several application domains. We have some ideas on how to allow additional flexibility in our solution. Another relevant aspect to be tackled in the future is that the proposed protocol does not feature security mechanisms, there are several aspects to be considered, namely the fact that processes can impersonate other processes issuing ALIVE messages in their name, an aspect related with identity verification in decentralised systems that we will tackle in the future by taking advantage of asymmetric cryptography to manage identities. The other evident attack vector is that a malicious node could disseminate SUSPECT messages for another node using an arbitrary timestamp associated with that identifier. This can be mitigated by having identifiers be signed by the node itself, which is feasible since only a process can increase the timestamp associated with its own identifier. This however still allows malicious or faulty processes to incorrectly disseminate SUSPECT messages using valid identifiers received through ALIVE messages. This must be tackled by further research in the future.

Finally, an aspect that will be addressed in future work is the experimental evaluation of this solution on different settings, as to measure the communication and computational overhead, and ascertain in which conditions can the protocols become stable (i.e., where in a steady state the local perception of nodes regarding elements in the membership remains unchanged).

### 2.2.3 Integrating the Actyx middleware: reliable and durable event broadcast

The Actyx middleware [101] is a proven software product for the automation of non-real time high-level workflows on the factory shop floor. It became clear within the first six months of this project that Actyx would be a suitable tool to be added to the TaRDIS toolbox. Initially it was proprietary software owned by the project partner ACT, commercially licensed to its customers, which includes both factories and factory automation software providers. To more effectively support TaRDIS, ACT has released Actyx under the Apache 2 open-source licence in October 2023, with the source code being available on GitHub.

#### 2.2.3.1 Overview of the Actyx middleware pre-TaRDIS

The deployed artefact of Actyx is comparable to a database: the application uses the Actyx SDK to communicate with an external Actyx process to publish events for dissemination among the other Actyx nodes, and to query the locally stored events, which includes events published on other nodes. The events are labelled with a set of tags at the time of publishing, allowing queries to select specific substreams of events pertaining to any given item or workflow that is modelled on top of this system—this can be compared to the topics in a pub–sub system like Kafka or Redis, with the difference that Actyx permits an event to carry multiple tags and gives queries the freedom to select arbitrary combinations of tags.

There are Actyx SDK implementations for the Typescript and Rust programming languages, with Typescript being the preferred language by ACT's customers—this notably includes the customer with which the TaRDIS evaluation use case will be implemented. Most communication patterns in this use case are expressed using the Actyx machine-runner library [102] that is built atop the Typescript SDK and stands to be extended by included into the TaRDIS toolbox as part of work packages 3 and 4—it implements the workflow-based communication abstraction described in deliverable D3.1.

Actyx itself is implemented using the Rust language, with data stored using sqlite3 and swarm communication performed using the libp2p library. The stored data structures are persistent trees with a large branching factor and summary information regarding timestamps and tags at every level—this allows quick and efficient access to exactly those events that are selected by a given application query. The tree nodes are content-addressed and sent between Actyx nodes using the IPFS [103] suite of protocols, which nicely matches the append-only log structure used to realise durable and reliable delivery of events across the whole swarm. Updates regarding the event log contents are sent to other swarm nodes using the gossip subprotocol that comes with libp2p by regularly advertising the current root node's hash of the local node's event log tree. Added resilience and indirect event dissemination is achieved by also advertising the root hashes known by the local host for other nodes' trees. This design has proven to be extremely resilient in factory applications.

The Actyx system does not model different access or security rights for each swarm node, the application in factory use cases usually does not require such multi-tenancy since all participating edge devices are owned and operated by the same commercial entity and are protected (physically and virtually) by the factory IT department according to the factory's needs. In other words, an Actyx swarm forms a single trust and security domain—on the Actyx level this is expressed by a shared secret (a pre-shared cryptographic key) known to all devices belonging to a given deployment.

#### 2.2.3.2 Adaptations required for Actyx integration into TaRDIS

The Actyx middleware described above provides a solid foundation on which a TaRDIS tool will be built, but it does require adaptations, which fall into the following broad categories:

- internal modifications to make it possible to use the other results described in this document in the implementation of the Actyx services;
- modifications of the external presentation of Actyx to the developer to conform to the TaRDIS APIs, models, and development methodology.

In the first group fall refactoring's of the internal component structure, decoupling the current libp2p usage from the core Actyx abstractions (i.e. the append-only log and local query engine). New interfaces need to be created and installed for swarm membership (joining, monitoring, querying), update dissemination (broadcasting log summaries), and event data exchange (synchronising log changes between pairs of nodes).

The second group contains changes to the offered API that range from cosmetic name changes to the addition of new facilities (e.g. for security aspects like key management). It also contains the larger effort of making it possible to use Actyx as a library instead of as a separate process. This is necessary to reliably provide the communication services to apps on smartphones, which are notorious for denying background processes to keep running to conserve battery. We will achieve this by restructuring the code modules into a dynamically linked library with a C interface and native bindings for the languages used by TaRDIS use cases that employ Actyx (candidates are Typescript/JavaScript, Kotlin/Java, and Python).

#### 2.2.3.3 Current state and future work

The effort expended so far has been used to get the Actyx codebase on GitHub into shape for open sourcing, including setting up the CI pipeline on GitHub actions (as is customary in the relevant developer communities). We then changed the internal module structure such that it can be published on the standard Rust software repository https://crates.io/ without undue difficulty (previously the codebase consisted of a large number of separate crates with complex interdependencies, as open-source publishing had been no design constraint). The result is that the Actyx service binary can be installed using `cargo install ax` as is expected from a Rust tool. The main functionality is bundled in the `ax-core` crate, which is used as a library from the `ax` crate, albeit without finalised API design and separation of concerns. This work is ongoing and remains to be completed before external developers can effectively embed Actyx in their own applications.

The work of installing TaRDIS interfaces within Actyx and presenting TaRDIS APIs to the exterior has not yet begun, as this very deliverable is part of the required specification guiding such efforts.

## 2.3 Decentralised Data Management and Replication (Task 6.2)

Task 6.2 focuses on developing the data management support for building TaRDIS platform and applications. In this context, several challenges have been identified.

First, TaRDIS intends to support heterogeneous settings, composed of nodes with different resources, ranging from powerful cloud nodes to small client devices and edge nodes. Furthermore, some nodes of the system may experience limited connectivity during some periods. For addressing this challenge, it is necessary to develop data replication algorithms that adapt to these heterogeneous settings.

Second, TaRDIS intends to support dynamic settings, where nodes' interests and location can change over time. To address this challenge, it is necessary to develop partial replication algorithms, where each node only stores part of the data. Furthermore, these algorithms need to support dynamic replication, as the objects that need to be replicated at each node will vary over time.

Finally, TaRDIS intends to support nodes with Byzantine behaviour – this behaviour can be the result of software bugs, node malfunction or a security breach in which a node becomes under the control of an attacker. For addressing this challenge, we will need to develop algorithms for Byzantine fault tolerance.

During the first year of the TaRDIS project, the work in this task has focused mostly on the first two challenges. The main results produced are the following:

- Section 2.3.1presents Arboreal a data management system for supporting edge computing, in which applications move from the cloud to the edge of the network.

Arboreal is designed for being deployed in a system that includes a wide variety of devices, from cloud nodes to edge nodes with different levels of resources and client devices. Besides supporting heterogeneous settings, Arboreal replication algorithms are also designed to support dynamic partial replication to support client mobility.

- Section 2.3.2 presents the status of PotionDB a data management system designed to be deployed in a small number of nodes, potential geo-distributed, and with support for partial replication. Unlike Arboreal, PotionDB supports a transactional API, thus providing a more powerful API for the application. Still under development, it is the support for materialised views over geo-partitioned data, providing a mechanism for supporting recurrent queries that are common in applications.

- Section 2.3.3reports on the initial effort for allowing the integration of third-party storage solutions in the TaRDIS ecosystem.

### 2.3.1 Arboreal: Extending Data management from Cloud to Edge leveraging Dynamic Replication

Computing infrastructures have been extending from the cloud to the edge to support low response times and high availability. By bringing computations closer to clients, edge computing addresses the demands in everyday services, such as social media or online shopping, while enabling novel latency-critical services, such as location-based games, autonomous vehicles [74], and live video analytics [75].

Deploying application logic on edge nodes has limited benefits if requests require fetching data from the cloud. However, extending cloud data storage solutions to address the challenges of the edge is non-trivial. Edge nodes often have limited resources, storing only a subset of an application's data. Combined with dynamic data needs at each edge location, this renders traditional data partitioning techniques unsuitable. Additionally, the significantly larger number of edge locations and their higher susceptibility to failures require a scalable data replication solution capable of handling entire edge location failures. Addressing these challenges to enable fully-fledged applications at the edge requires new data replication solutions tailored specifically for this environment.

We have been working in Arboreal, a novel distributed data management system with a decentralised data replication protocol designed specifically for edge environments. The replication protocol dynamically replicates data across edge locations, enabling applications to be deployed at the edge with complete local data access, resembling a cloud deployment. Simultaneously, it ensures global causal+ data consistency, preventing data anomalies. We now detail Arboreal.

#### 2.3.1.1 Towards Stateful Edge Applications

To fully leverage the potential benefits of edge computing, providing low latency and reducing centralised component loads, application requests need to be fully processed in client nodes. To this end, the application logic on edge nodes needs to have read and write access to local replicas of application data. To support this stateful edge application model, storage systems must extend from data centres to edge locations, addressing the following challenges, distinct from those faced by traditional cloud storage systems [63,60,62]:

**Partial and dynamic replication.** Applications with large user bases are expected to leverage a considerable number of edge locations. These edge locations, characterised by less powerful and reliable computational resources compared to core data centres, pose unique challenges, as it is imperative for application components to have unrestricted data access while maintaining consistent guarantees, regardless of executing in the cloud or at the edge. Achieving this requires a data storage solution supporting fine-grained *partial and dynamic replication*. This means that the set of data objects replicated at each edge location evolve over time to reflect the access patterns of clients accessing the applications in that location. The underlying replication protocols must dynamically adapt to ensure timely propagation of data updates to the correct locations, preventing components and clients from encountering stale data, and ensuring operations become visible across the system.

**Scalable consistency.** To maintain application logic mostly unchanged across different execution locations, ensuring some form of data consistency is crucial. While strong consistency simplifies application logic by avoiding data anomalies, it proves impractical for edge settings due to latency and availability issues arising from coordinating numerous replicas. In edge environments, it is more suitable to rely on a weak consistency model, which relaxes consistency for better availability and response times. To mitigate anomalies in eventual consistency models, many solutions adopt *causal+ consistency* [64,67] which provides the strongest highly available possible consistency guarantees.

Causal+ consistency is based on the *happens-before* [67] relationship, where a write operation can only be made visible in a replica after all writes that causally precede it. Concurrent writes can be made visible in any order, as long as all replicas eventually converge to the same state. The key challenge lies in tracking the *happens-before* relationship among operations. Common approaches use vector clocks [68], but scalability is limited as metadata grows linearly with the number of replicas. Other approaches, like tree-based topologies [76], avoid growing metadata costs but are sensitive to changes in the replica set, having weak fault tolerance. Some solutions [77] rely on centralised components, limiting the benefits of using multiple edge locations. This work presents a scalable solution for tracking causal dependencies across write operations that can *scale to hundreds of locations*, in a context involving dynamically replicated data objects, and in a way that can deal with frequent replica set changes and be fault tolerant.

**Client mobility.** Applications benefiting most from the envisioned stateful edge applications involve users dispersed across different locations. Examples include collaborative applications (e.g. Google Docs), autonomous vehicles, smart-city applications, multiplayer mobile games, or stateful serverless computing [78]. In these applications, where low response times are crucial, users may change locations while using the application. When doing so, users should be able to migrate from interacting with an application component on one edge location to a closer one without data consistency anomalies. This emphasises the need for a distributed data storage solution capable of handling *mobile clients* without compromising consistency guarantees.

Next, we present the design of Arboreal which, to the best of our knowledge, is the first distributed storage system providing *causal+ consistency* while supporting fine-grained *partial and dynamic replication*. It can *scale to hundreds of different locations*, is *fault-tolerant*, and effectively supports *mobile clients*. This unique combination of features makes Arboreal especially well-suited for deployment in edge environments.

#### 2.3.1.2 Arboreal Design

Arboreal is a distributed data management system designed for the edge, featuring a novel scalable replication protocol that ensures causal+ consistency. Scalable to hundreds of edge locations, Arboreal seamlessly adapts to membership changes and effectively manages faults and network partitions. Arboreal is designed to support extending cloud applications to edge environments, employing dynamic partial replication. This enables edge nodes to automatically adjust the set of locally replicated data objects in response to changes in client access patterns while allowing clients to move across edge locations without compromising consistency guarantees. All of this is achieved in a fully decentralised manner.

**System model:** Our solution assumes a set of cloud data centres spread across different geographic regions, in which a distributed (geo-replicated) NoSQL database is deployed. Deployment specifics of this database (e.g., replication protocol, partitioning scheme) are orthogonal to this work. We consider a set of *edge locations* equipped with computational resources. Arboreal extends the database from the cloud data centres to edge locations within respective regions. For this, an instance of Arboreal is deployed both in each data centre and each edge location. We assume that an edge location may consist of one or multiple edge nodes, however, it is always treated as a single node, with a single instance of Arboreal being deployed in each edge location. No assumptions are made about replication and data partitioning schemes within each edge location, focusing instead on data replication across edge locations. To accommodate diverse scenarios and applications, we assume that, at any time, Arboreal can be dynamically deployed, along with application components, in new edge locations and that edge locations may fail, or Arboreal may be decommissioned from them. Applications deployed on edge nodes rely on Arboreal for consistent local data access to execute client operations. Clients, potentially mobile, can migrate between edge locations at any time, and have dynamic workloads, with the set of accessed data objects possibly changing over time.

**Data Model:** Arboreal offers a key-value store interface, akin to other highly available distributed databases [63], where each data object is identified by a unique key. Client's issue read or write operations on data objects without constraints, and Arboreal ensures that: (1) data objects are replicated transparently to the edge locations where they are accessed; (2) write operations are propagated to all locations currently replicating the modified data object; and (3) clients always observe a state respecting the causal order of operations. Arboreal makes no assumptions about how data is stored in each node. For convergence, a *last-writer-wins* policy is used, relying on operation (logical) timestamps.

#### *2.3.1.2.1 Replication*

As discussed earlier, a key challenge of this work is overcoming limitations in replication protocols providing causal+ consistency, in a way that is suitable for a large-scale edge environment. To address this challenge, it is crucial to enable edge locations to synchronise (propagate operations and data objects) directly with each other while simultaneously ensuring that metadata, essential for enforcing causality and supporting (object-grained) dynamic replication, does not grow linearly with the number of edge locations.

##### *2.3.1.2.1.1 Hierarchical Approach*

For this, Arboreal employs a hierarchical design, with each edge location hosting an instance of Arboreal. These edge locations define a tree structure rooted at their regional data centre, establishing the region's *control tree*. Figure 2.3.1.1 shows a geo-distributed example of an Arboreal deployment with three data centres, each having its *control tree*. The management of the *control trees* is fully decentralised, with nodes communicating solely with their parent node and children. Nodes only retain detailed information about their children and minimal information about their ancestors (i.e., nodes in the path between itself and the root of the *control tree*). This decentralised structure allows for latency-sensitive tasks, such as failure recovery, mobile client handling, and creating replicas of data objects to be performed in a localised fashion, involving as few nodes as possible.

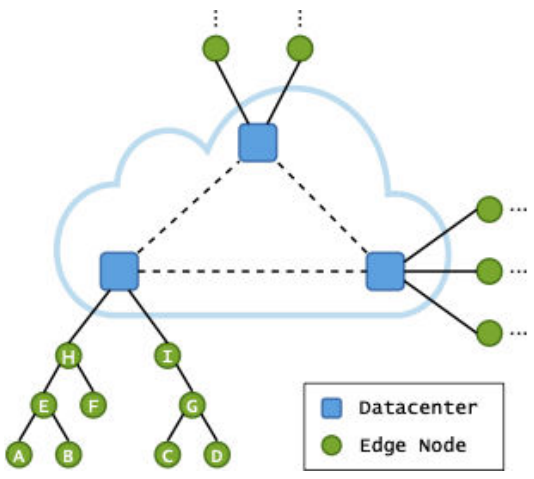


*Figure 2.3.1.1 - Design of Arboreal, with 3 geographic regions.*

To establish the *control tree*, when an instance of Arboreal is deployed on an edge location, it uses a heuristic to connect to the most suitable existing instance in the *control tree* of its region. Note that the goal of Arboreal is to replicate data to applications running on edge locations and is not an orchestrator that decides where and when to deploy the application. To accommodate diverse edge scenarios, both the heuristic defining the *control tree* and the information used by the heuristic are configurable by the application developer.

Given the significance of geographic locality in edge computing, our Arboreal implementation employs geographic distance between edge locations as the primary metric for forming the *control tree*. However, depending on the application, various metrics can be used, such as latency between edge nodes, client locations, or even predicting future demand for the application.

#### 2.3.1.2.1.2 Enforcing causality

For tracking causal dependencies between operations, vector clocks are commonly used, but their cost is prohibitive for large-scale systems, especially with partial replication. Even solutions that attempt to minimise this cost require, in the worst case, metadata that grows linearly with either the number of partitions or data centres [68]. Thus, we need a solution providing causal+ consistency using metadata that does not grow linearly with either.

**Causal Dissemination:** We start by leveraging the hierarchical topology of Arboreal, which allows achieving causal consistency without requiring metadata [76]. For this, nodes form the *control tree* by establishing FIFO channels to their parent node and children. When a node receives a write operation from a channel (i.e., from a parent or child), it atomically executes the operation locally and puts it on the outgoing queue of every other channel. Additionally, local operations from clients are atomically added to the outgoing queues of all channels. This ensures operations are always enqueued (and thus, executed) after all their causal dependencies. While ensuring causal consistency, this approach assumes clients always issue operations to the same node and a static *control tree*, which are unrealistic assumptions for the edge.

**Timestamping:** To overcome these limitations, Arboreal additionally utilises Hybrid Logical Clocks (HLCs) [79] to enforce causal consistency. HLCs combine physical time for monotonic advancement in each node with logical clocks for capturing causal relationships between operations despite physical clock anomalies. When a client's write operation is received by a node, it is tagged with a timestamp from the local HLC. This timestamp is propagated with the operation through the tree and is stored with the object data in each node. Additionally, Arboreal employs the notion of *Branch Stable Time* (BST). A BST is computed individually by each node as the minimum between its current HLC time and the BST of each child. The BST captures that no operation with a lower timestamp will be generated by any node in its branch (a branch consists of the node itself and its descendants in the *control tree*). Nodes periodically propagate their BST to parents and children, including the BST of all ancestors when propagating to children. This ensures each node tracks the BST of its children and all ancestors. Combined with the *causal dissemination* technique, BSTs allow Arboreal to provide causal consistency in every scenario, including failure recovery (as explained later).

###### *2.3.1.2.1.3 Dynamic and Partial Data Replication*

An essential aspect of edge computing is that edge locations cannot be expected to have resources to replicate the entire dataset of an application. As such, partial replication becomes a key aspect of Arboreal. Moreover, to support a wide range of applications, Arboreal must adapt not only to changes in client access patterns but also to mobile clients that can change the connected edge location at any time. Unlike cloud-based data management systems that typically use static data partitions, Arboreal needs to allow edge nodes to dynamically change the set of replicated data objects at any time with fine granularity. However, keeping track of which nodes replicate which data objects across a large-scale system can be costly and require substantial metadata propagation, especially with dynamic sets of nodes and data objects. To address this challenge, we rely on the hierarchical topology.

In Arboreal, each data object is individually replicated to a subset of edge nodes. This is done in a way that ensures any node always contains the data objects its children replicate, resulting in each data object forming a subtree of the *control tree*, which we refer to as the *replication tree* of an object. Figure 2.3.1.2 illustrates the evolution of an Arboreal deployment with two objects replicated on different sets of edge nodes that change over time.

While this restriction in data object replication may seem limiting, forcing edge nodes to replicate data objects that their current clients may not be interested in, it is actually a beneficial design decision for three reasons: (1) as a subtree of the *control tree*, an object's *replication tree* inherits causal dissemination guarantees, providing causal+ consistency globally across all objects; (2) when the *control tree* is repaired after node failures, the *replication trees* involving the faulty nodes are also repaired, enabling the system to quickly recover from failures with minimal client impact; and (3) it aids client mobility. When a client migrates to another node, even if that node does not replicate the required data objects, there is a high chance that one of its close ancestors does, allowing the client to quickly resume its operation. This design ensures *replication trees* form in a decentralised manner based on client needs. Each node only needs to track the objects it replicates and the objects each of its children replicates. This mechanism assumes that storage capacity increases moving up the *control tree*, closer to cloud data centres, which we believe to be a reasonable assumption for an edge environment.

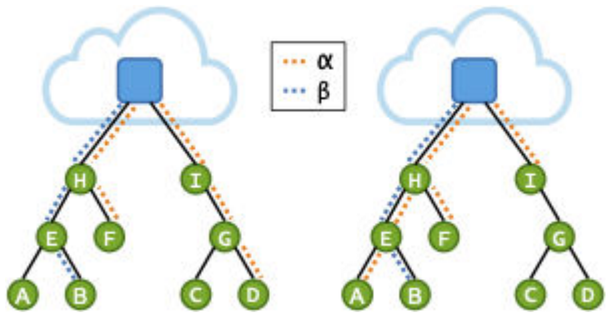


*Figure 2.3.1.2 - Dynamic partial replication in Arboreal.*

**Replica creation:** When a client requests a data object not replicated in its connected edge node, that node sends a request to its parent node, asking to be added to the object's *replication tree*. If the parent node is part of it, it sends the current object version to the child node and keeps track that this child node is now replicating the object. If the parent node is not part of the *replication tree*, it forwards the request to its own parent, and so on, until the request reaches a node replicating the object (or the cloud data centre, replicating all objects). This node then sends the object to its child node, and the process repeats until the object reaches the node that initially requested it. This process is illustrated in the figure above, where node A (and consequently, node E) is added to the *replication tree* of a.

**Garbage Collection:** Due to the possible large number of data objects and limited storage in edge nodes, Arboreal employs a garbage collection process. Each node tracks the last time each data object was accessed and periodically removes objects not accessed for a configurable duration. When removing a data object, a node informs its parent that it no longer replicates that object. Nodes can only garbage collect objects not replicated to any children to prevent breaking the *replication tree*.  Figure 2.3.1.2 shows object a being garbage collected in nodes D and G, as their clients are no longer accessing it, being removed from its *replication tree*.

The decentralised design of Arboreal's replication protocol allows it to scale to a large number of edge nodes while supporting fine-grained replication of data objects. This allows each node to only track metadata proportional to the number of objects it replicates, avoiding linear growth with the total number of edge nodes and data partitions/objects.

#### *2.3.1.2.2 Fault Tolerance*

Unlike cloud environments, where individual nodes can fail but it is unlikely that an entire data centre does, in edge environments, we need to assume that entire edge locations can fail or become partitioned at any time. Therefore, Arboreal must not only be capable of recovering from failures but also provide data persistence guarantees when they occur.

##### *2.3.1.2.2.1 Data Persistence*

Arboreal provides a mechanism allowing applications using it to specify the *persistence level* of write operations. Effectively, this allows applications to specify how many nodes upstream in the *replication tree* a write operation must reach before it is considered to be persisted. This mechanism is especially beneficial for applications requiring data persistence guarantees in more volatile edge locations. Its design prevents scalability issues by avoiding extra communication steps between nodes and the need for nodes to track the origin of each operation.

**Persistence ID:** Before forwarding a local write operation to its parent, a node assigns a *persistence ID* to the operation. Upon receiving a write operation from a child, a node assigns its own persistence ID to the operation, mapping it to the child's persistence ID, and then propagates the operation to its parent. The persistence ID is essentially a local counter for each node, incremented whenever a node assigns it to an operation. The left side of Figure 2.3.1.3 illustrates this mechanism in action. A client issued 3 write operations in node A, with the first 2 reaching the data centre, and the last one only reaching node E. Additionally, an operation in node F reached the data centre. The figure depicts the mappings by each intermediate node. For example, the data centre operation with persistence ID H3 originated in node A with persistence ID A2. Importantly, nodes lack information about the origin of each operation, enabling Arboreal to scale by avoiding storing metadata for a large number of nodes.

**Persistence level notifications:** Periodically, each node communicates to its children the persistence level of their operations through a list of pairs (persistence ID, persistenceLevel). Each pair signifies that all child operations up to persistence ID have been persisted in *persistenceLevel* nodes. Upon receiving this list from its parent, a node maps the persistence ID of each pair to the persistence ID of the child, increments the persistenceLevel of each pair by one, and, if operations from the child are missing, adds an entry with the highest of those operations and a persistenceLevel of 1. The persistenceLevel of operations reaching the data centre is conveyed as infinite. This mechanism is depicted in Figure 2.3.1.3, where node A receives acknowledgment that its operations up to A2 have been persisted in the data centre and A3 in one level above it. This persistence level information is periodically sent to children piggybacked on the BST messages. Upon receiving confirmation that an operation has been persisted in the data centre, nodes can forget all persistence information related to that operation. Regardless of the requested persistence level, this mechanism is always active for all operations to ensure no loss of operations during fault recovery.

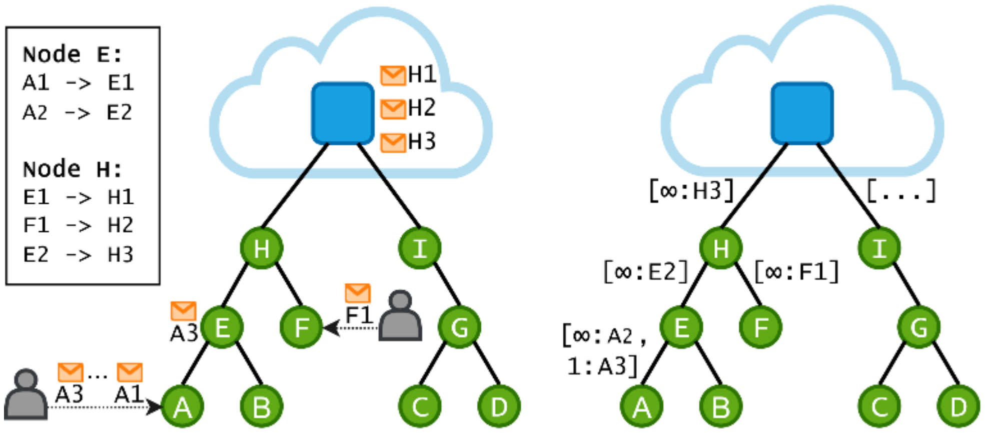


Figure 2.3.1.3 - Data persistence in Arboreal.

### 2.3.1.2.2.2 *Fault Handling and Recovery*

Due to the nature of edge environments, decentralised fault handling and recovery are essential for Arboreal. The main challenges involve: (1) rebuilding the control tree after node failures; (2) ensuring consistency during the rebuilding process; and (3) maintaining data persistence through failures and reconfiguration.

To address the first challenge, each node, being aware of all its ancestors, can independently rebuild the control tree after detecting a failure. Nodes attempt to connect to their grandparents if their parents fail, falling back to the great-grandparent and so on, until reaching the data centre. Since every ancestor of a node replicates a superset of its data, this simple approach automatically repairs not only the control tree, but also any disconnected replication trees.

For the second challenge, when connecting to a new parent, Arboreal employs a 3-step protocol to synchronise with its new parent, ensuring no violations of consistency guarantees:

(1) The (to-be) child node sends a *Sync Request* to the (to-be) parent node, including its current BST and a list of replicated objects with associated timestamps;

(2) The parent node registers the child as a new child, checks if it has any outdated objects, and replies with a *Sync Response* containing the child's new ancestor list, BSTs, and a list of updates for the child's outdated objects;

(3) The child updates its ancestor list and BSTs and instals the outdated objects. It sends the parent requests for any pending replica creation requests and client write operations. Finally, the child propagates a *Reconfiguration Message* to its children, containing their new ancestor list and BSTs, which is propagated to its entire branch.

To address the third challenge, after the synchronisation, both the child and every node in its branch re-propagate local write operations with pending persistence requests, as the persistence mechanism may break down during reconfiguration.

Though the synchronisation protocol may seem complex, it allows each node to reconnect itself to the *control tree* without centralised coordination. This decentralised approach enables Arboreal to recover from multiple failures in parallel.

Regarding clients, failures can affect them in two ways:

(1) If a client's connected node remains operational, but one of its ancestors fails, the client can continue normal operation. The only noticeable effect is that persistence confirmations may be delayed as they will only arrive once the current node reconnects to the *control tree*;

(2) If the node to which the client is connected fails, the client must reconnect to a new node. Operations with persistence confirmation are guaranteed to be visible in the new node, but those without may have been lost. The client can then re-execute lost operations or perform read operations to verify the persistence of those operations.

### 2.3.1.3 Client Mobility

In edge application scenarios with mobile clients, such as users with smartphones, Arboreal must seamlessly support clients moving from an edge node to a closer one while maintaining consistency guarantees. Notice that in this context, clients can attach to the closest edge location by taking advantage of DNS services that reply with the closest instance, or by being explicitly redirected by other nodes in the system. This is a non-trivial task as the new node may not have any information about the client's previous node (which may even have failed). Therefore, we assume that the nodes involved in this procedure are unable to communicate, with the client storing all required information.

**Client state:** Clients of Arboreal track two pieces of metadata: (1) the list of ancestors of its current node; (2) a timestamp with its current causal dependencies. This timestamp is updated upon receiving responses to operations, and always contains the highest timestamp seen. Read operations return the timestamp of the object read, while write operations return the timestamp assigned to them by the client's node. Figure 2.3.1.4 shows the BST of nodes and the timestamp and list of ancestors of a client in an example deployment.

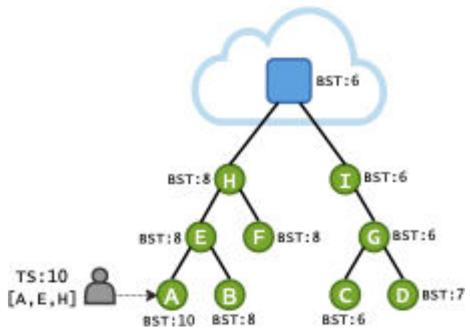


*Figure 2.3.1.4 - Client mobility in Arboreal.*

**Mobility procedure**: The combination of the client-side timestamp and list of ancestors with the node-side BST enables Arboreal to support quick client migrations without compromising causality. When a client connects to a new node, it sends a migration request to the new node, containing its current timestamp and list of ancestors. The new node then compares the received list with its own list of ancestors, leading to one of the following cases:

(1) If the new node was not in the client's list of ancestors, signifying a *horizontal migration* to a new branch of the control tree}, the new node identifies its closest ancestor that is also in the client's list of ancestors. It responds to the client only after receiving a BST from that ancestor greater than the client's timestamp. This ensures that the new node responds to the client only when it is certain that it has observed all operations the client depends on. In Figure 2.3.1.4, if the client migrates to node F, it waits until F receives a BST of 10 from H (the closest common ancestor).

(2) If the new node was in the client's list of ancestors, indicating a *vertical migration*, the new node identifies which of its children is an ancestor of the client's old node (or the node itself). It responds to the client once it receives a BST from that child that is equal or greater than the client's timestamp. In Figure 2.3.1.4, if the client migrates to E, it waits until E receives a BST from A with at least 10 (which should be quick, as the BST of A is already 10). In cases where the client migrates to the parent of a failed node, the new node immediately accepts the migration as there is nothing to wait.

After this process, the client receives an updated list of ancestors. The metadata stored in the client only needs to be readable by Arboreal and may be opaque (e.g., encrypted) to the client itself, preventing information leakage about the internal organisation of the system.

The duration of the migration process increases as the client moves farther from its old node, requiring the new node to wait for a BST from a more distant node in the *control tree*. However, in typical scenarios, clients migrate to close-by nodes, resulting in swift migrations.

This mechanism might be overly cautious. For instance, if the client aims to migrate to F, it depends on the BST of H, which, in turn, relies on the BST of B. However, node B might not have participated in any operation observed by the client, causing potential delays in migration completion. While we recognize this cautious approach may introduce unnecessary delays, alternatives that accelerate migrations typically involve additional metadata or explicit migration messages sent through the tree. Such approaches could compromise Arboreal's scalability and fault tolerance.

#### 2.3.1.4 Evaluation

In this section, we study the performance of a prototype of Arboreal using an edge environment setup. We compare Arboreal with other solutions that provide causal+ consistency on the edge, namely a decentralised solution using vector clocks (Engage [80]) and solutions using a centralised topology to enforce causality (Colony [70]). For the former we use the provided code, while for the latter, as the code is not available, we mimic their centralised topology by implementing a version of Arboreal, named *centralised*, where all edge locations connect directly to the data centre. We also compare Arboreal against Cassandra [81], a cloud database that due to its configurable partial replication and peer-to-peer synchronisation can be used in edge settings.

##### *2.3.1.4.1 Experimental Setup*

We conducted experiments in a cluster of 10 machines, each having 2 AMD EPYC 7343 processors with 64 threads and 128 GB of memory, connected by a 20 Gbps network. We deployed a docker swarm across all machines, with an overlay network connecting all containers. A container with no resource restrictions on one machine serves as the data centre, while up to 200 containers distributed across the others, with resources restricted to 2 virtual cores and 4 GB of memory, represent edge nodes. Each container executes an instance of Arboreal.

To emulate a geographic region, we randomly distributed the 200 nodes across a virtual 2-dimensional space, representing edge locations, with the data centre at the center. We used Linux *tc* to emulate latency between nodes based on their Euclidean distance, with a maximum latency of 150ms from an edge node to the data centre. Experiments were conducted 3 times on 3 such distributions using either 20 or all 200 nodes. Figure 2.3.1.5 shows an example distribution with the formed *control tree*, using the *deep* layout. Additionally, we deployed 200 client containers, distributing them across the same virtual space and setting the latency between each client and edge node using the same method, with a minimum latency of 10ms.

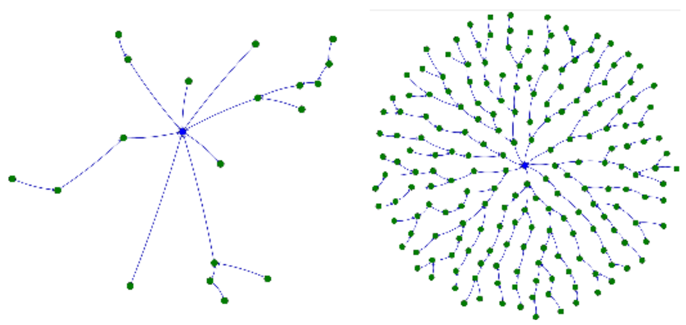


*Figure 2.3.1.5 - Example node distributions in a geographic region.*

### 2.3.1.4.2 Performance

In our performance benchmarks, we evaluate the performance of Arboreal in terms of operation throughput and visibility times, comparing it against other causal+ solutions, Cassandra, and using different *control tree* layouts.

In these experiments, clients connect to their closest edge node and perform operations on data objects based on their location. The geographic region is divided into 8 equal segments, with each being assigned a data partition. Clients perform operations on data objects in their segment and the 2 adjacent segments. This setup assesses the performance of Arboreal with data locality, where clients are more likely to access nearby data objects. As the replication of data in Arboreal is dynamic and based on client access patterns, this results in each edge node replicating data objects from at least 3 partitions, with the data centre replicating data objects from all 8 partitions.

For Cassandra, as partial replication is based on a static placement, we configured each edge node as an independent cluster (*datacenter* in Cassandra terminology) and created partitions (*keyspaces* in Cassandra terminology) so that each edge node replicates all data objects from the 3 partitions accessed by clients, while the data centre replicates all 8. A similar approach was used to distribute partitions in Engage.

#### 2.3.1.4.2.1 Throughput

Figure 2.3.1.6 shows the throughput of Arboreal compared to Engage and a centralised topology solution, varying the number of nodes and the number of distinct data partitions. We show the throughput of write operations only, as read operations execute locally in all solutions. Due to weak consistency allowing nodes to respond to clients without coordination with other nodes, measuring throughput on the clients is unreliable, as operations may be processed in their local nodes at a higher rate than they are replicated to other nodes. As such, the values displayed represent the maximum throughput measured in the data centre node for each solution.

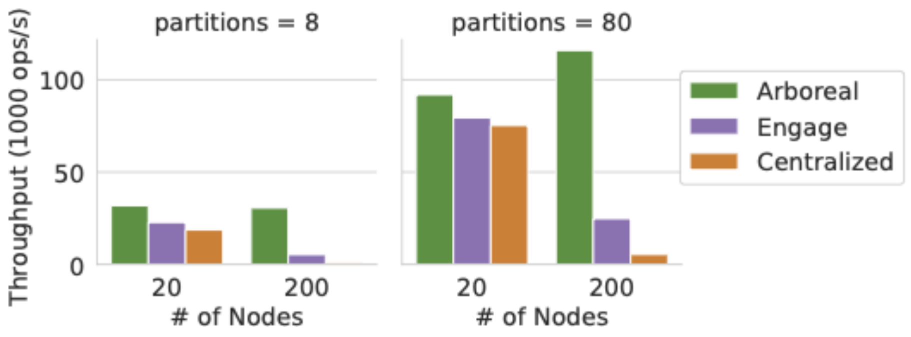


*Figure 2.3.1.6 - Throughput of Arboreal versus causal+ solutions.*

Two main conclusions can be drawn from these results: (1) Unlike existing causal+ solutions, Arboreal scales to hundreds of nodes without performance degradation. This is a result of avoiding both vector clocks (used in Engage) and centralised topologies (as the ones used in [70]) for causality enforcement, opting for a decentralised hierarchical topology. Despite the expectation that a seemingly infinitely resourced data centre could handle a centralised solution with a large number of edge nodes, this is not the case, as causality enforcement requires some form of (partial) serialisation of operations, limiting parallelism; (2) Increasing the number of data partitions (by specialising data objects accessed by clients) and nodes (increasing the number of replicas for each data object) increases the throughput of Arboreal. This happens since each added node removes load from existing ones, allowing more operations to be processed in parallel. This is a key advantage of Arboreal's dynamic replication mechanism, distinguishing it from state-of-the-art solutions.

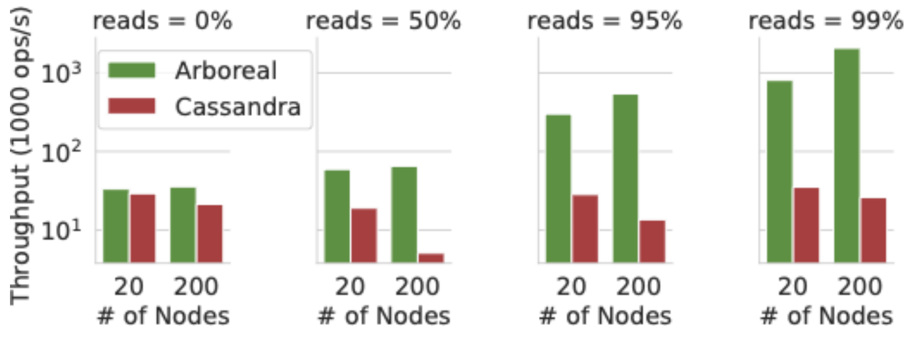


*Figure 2.3.1.7 - Throughput of Arboreal versus Cassandra.*

Figure 2.3.1.7 shows throughput for Arboreal and Cassandra as observed by clients, varying the number of nodes and the read/write ratio. Note the Y axis is in log scale. For more reliable measurements, we use infinity persistence in Arboreal and *quorum* consistency in Cassandra, ensuring that writes are only acknowledged to clients after being replicated to the data center and a majority of edge nodes, respectively, preventing artificial throughput inflation. The *quorum* consistency level in Cassandra also ensures that clients observe the latest value of a data object, which provides some consistency guarantees (although different from Arboreal, as it can violate causality), making the comparison with Arboreal fairer.

In write-only scenarios (0% reads), Arboreal achieves higher throughput by leveraging its hierarchical topology and persistence mechanism. Nodes that Cassandra must send each write operation to all other nodes replicating the data object, then await acknowledgement from a quorum. In contrast, Arboreal sends write operations only to the parent (and any children replicating the object) and waits for the persistence acknowledgement. This greatly reduces message complexity, allowing higher throughput. With increased read operations, by processing reads locally, Arboreal outperforms Cassandra, which requires coordinating with a quorum before responding. While Cassandra can avoid coordination, sacrificing data consistency guarantees, to increase its throughput, Arboreal can process reads locally while providing causal+ consistency. Regardless, Figure 2.3.1.7 shows that the hierarchical topology of Arboreal is much better suited for edge environments than traditional solutions designed for cloud environments.

###### 2.3.1.4.2.2 Visibility Times and Tree Layouts

In this section, we explore the impact of different control tree layouts in Arboreal's hierarchical topology on the visibility times of operations and compare them with a centralised topology.

Using the same setup as the previous experiment, with 200 nodes and only write operations, Figure 2.3.1.8 presents the results through a boxplot, where each box shows the distribution of the visibility times of operations in the different topologies. *Arboreal deep* and *Arboreal wide* represent tree layout where the first has deep branches and a small number of children per node and the latter has the depth limited to 4, with each node with a large number of children The figure depicts visibility times for the closest remote node (1), the 5th closest, and all nodes. The values for 1 and 5 are crucial in an edge environment as clients in geographical proximity, connected to different but close-by edge nodes, are likely to access the same data objects.

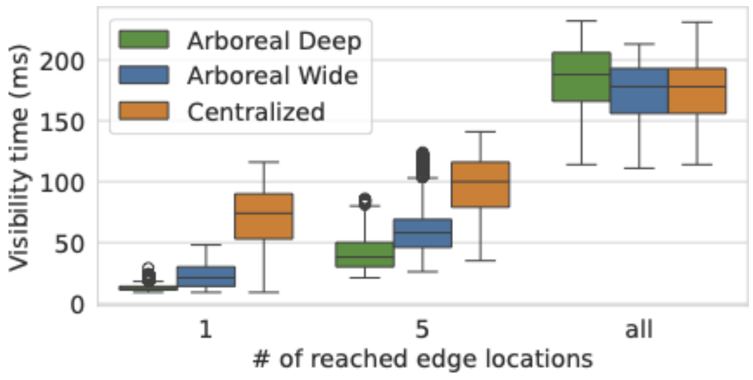


*Figure 2.3.1.8 - Visibility times of different topologies.*

Results indicate that using a hierarchical topology with the *deep* layout is optimal for achieving low visibility times, enabling rapid operation propagation to nearby nodes. However, reaching all nodes requires traversing a significant number of hops, resulting in higher visibility times. In contrast, the centralised topology always requires propagating operations directly to the data centre, resulting in much higher visibility times. The hierarchical wide layout serves as a balanced compromise, enabling quick propagation to nearby nodes while matching the speed of the *centralised* topology in reaching *all* nodes. Overall, these findings show that a hierarchical topology is better suited for edge environments than a centralised one.

### 2.3.2 PotionDB: Strong Eventual Consistency under Partial Replication

Partial replication is important in different settings, from cloud computing to edge and peer-to-peer systems. In cloud computing, it is common to geo-replicate data for providing low latency to users spread across the globe. As both the data managed by these systems and the number of data centres increases, fully replicating data leads to problems. First, storing all data in all data centres imposes a large overhead in terms of storage and may be unnecessary, as some data is only needed at some geographic locations. Second, increasing the number of replicas makes replication more complex and costly, as each update needs to be propagated to all replicas.

In edge computing, an even larger number of edge nodes exist, making partial replication the only approach that can be used. The same happens in the case of peer-to-peer systems, in which it makes no sense to have all data in every replica.

Replicated databases [60,61,62] that provide strong consistency give the illusion that a single replica exists, requiring coordination among replicas for executing (update) operations. This leads to high latency and may compromise availability in the presence of network partitions. Databases that provide weak consistency [63,65] allow any replica to process a client request, leading to lower latency and high availability. Consequently, these databases expose temporary state divergence to clients, making it more difficult to program a system.

As discussed above, causality allows to avoid data anomalies derived from week consistency, which is relevant for developing applications, since the programmer does not need to define explicit logic to mask the effects of such anomalies when they happen. PotionDB is a geo-replicated memory key-value store with support for partial replication. PotionDB provides causal+ consistency, for improved latency and availability, with transactional causal consistency [68], for improved consistency. We note that while we have already presented a storage solution based on a distributed key-value store, PotionDB and Arboreal have significant differences that make them suitable to support different swarm applications. Arboreal extends the replication towards edge locations dynamically, and hence is more suitable for latency sensitive applications. PotionDB operates only across data centres, but provides support for transactions, therefore being more suitable for applications that need to manipulate several data objects in an atomic fashion.

#### 2.3.2.1 System Overview

PotionDB is a distributed database designed for supporting global services deployed at multiple locations. In these settings, (some) data items are only needed at some geographic locations. As a running example, we consider a large e-commerce site with online stores for different countries and clients spread across the world. The service includes data for products, with some products available only at some locations. The service maintains information about customers and their purchases, with clients being associated with one online store.

In this context, fully replicating the whole dataset might become too expensive. Additionally, as most data is tied to some geographic location, one can expect that the majority of accesses occur in that location. PotionDB adopts partial geo-replication, with data items being replicated only at some locations. This allows PotionDB to reduce replication cost when compared to solutions featuring full replication, saving on both storage, processing, and networking costs.

While the vast majority of application operations access data objects that are available at the local replica (e.g. user, product, or purchase objects), some operation may access objects that are not locally replicated (e.g. a user placing an order for a product that is not available in the local region).

#### 2.3.2.2 Data model

PotionDB is a distributed key-value database. The database stores a set of objects, and we define that the database state, DB, is composed of the set of objects stored by the database, DB = Objs. Objects are uniquely identified by a tuple *id = (key, bucket, type)*. Objects are stored in buckets, which are the unit of replication in PotionDB. Buckets are further logically grouped in containers. An object has a *key* inside the bucket.

PotionDB supports objects of different data types, including registers, counters, averages, sets and maps. Objects are implemented as CRDTs [69], guaranteeing that object replicas converge to a single state in the presence of concurrent updates.

#### 2.3.2.3 Interface

PotionDB offers a key-value transactional interface that we summarise in this section. An application issues interactive transactions, by executing `begin(clk)`, where clk is used to enforce causality between consecutive transactions.
A transaction proceeds with a sequence of operations: `get(txId, id)`, which returns the full state of the object; `read(txId, id,op)`, which returns the result of read-only operation op executed in the object; and `upsert(txId, id,op)`, which updates object id by executing operation op, or creates the object if it does not exist.

Operations defined in each object are type-specific - e.g. a set has a contains(e) operation to check if value e belongs to the set, and an `add(e)` and `remove(e)` to add or remove e from the set. A transaction ends with a `commit(txId)` for committing the transaction, making updates durable, or `rollback(txId)` to abort the transaction.

PotionDB also supports one-shot transactions, `oneShotTx(clk, (id, op)+)`, that include a sequence of read or write operations.

PotionDB's data definition API includes operations to create and delete buckets. Even if buckets have no associated data type, i.e., any object type can be stored in any bucket, we expect that applications store objects of the same type in each bucket. A document or a table row can be stored in PotionDB as a map CRDT, with each element of the map having its own type.

#### 2.3.2.4 Consistency

PotionDB is a weakly consistent database that provides Transactional Causal Consistency (TCC) semantics [68]. Intuitively, in TCC different replicas may execute transactions in different orders. A transaction accesses a causally consistent database snapshot taken in the replica where the transaction executes at the time the transaction starts. As in snapshot isolation [70], the snapshot reflects all updates of a transaction or none. Moreover, if a transaction *t* is included in the snapshot, all transactions that happened-before [67] *t* are also included in the snapshot. Unlike snapshot isolation, and similarly to parallel snapshot isolation [71], it is possible for two concurrent transactions to modify the same object, with updates being merged using CRDT rules.

For completeness, we now precisely define the guarantees provided by TCC.

We consider a database composed of a set of objects *O*. Each object is replicated in a subset of database replicas.A transaction $t_i$ is composed by a sequence of read and update operations to objects in the database. A database snapshot, $S_n$, is the state of the database after executing a sequence of transactions $t_1,\ldots,t_n$ in the initial database state, $S_{init}$, i.e., $S_n = t_n(\ldots(t_1(S_{init})))$. The set of transactions reflected in snapshot *S* is denoted by *Txn(S)*, e.g., $Txn(Sn) = t_1,\ldots,t_n$.

Under TCC, a transaction *t* executes initially in a database snapshot *S*. The transaction executes in isolation, independently of other transactions concurrently being executed. Thus, the result of a read operation, $r_i$, performed in transaction *t*, is obtained by executing $r_i$ against the state $u_j(\ldots (u_1(S))$, with $u_1,\ldots,u_j$ the sequence of update operations executed previously in *t*. After the transaction *t* commits, the sequence of updates of the transaction are applied in the relevant replicas (as the database is partially replicated, a replica may execute only a subset of the updates of the transaction).

We say that a transaction $t_a$ happened-before transaction $t_b$ executed initially in database snapshot $S_b$, $t_a < t_b$ iff $t_a$ *ÎTxn(Sb)*. Transaction $t_a$ and $t_b$ are concurrent, $t_a \| t_b$ iff not $t_a < t_b$ and not $t_b < t_a$ [67].

For an execution of a set of transactions *T*, the happens-before relation defines a partial order among transactions $T = (T,<)$. We say T' = (T,<') is a valid serialisation of $T = (T,<)$ if *T'* is a linear extension of *T*, i.e., <' is a total order compatible with <. Under TCC, only database snapshots that result from the execution of a valid serialisation of transactions to the initial database state can be used, i.e., a transaction is always executed in a causally consistent snapshot.

Transactions can execute concurrently, with each replica executing transactions according to a different valid serialisation. To guarantee state convergence, we use CRDTs [69,71], which guarantee that after executing the same set of transactions according to a valid serialisation, objects will have the same state (by relying on the deterministic conflict resolution policies defined in the CRDT used for each database object).

#### 2.3.2.5 Architecture

We designed PotionDB with partial geo-replication in mind. Thus, we assume PotionDB instances to be spread at different locations across the globe (Figure 2.3.2.1). Each location only replicates a subset of the whole data. The system administrator has control over where each object is replicated. This allows account for data locality to ensure fast access to data, while keeping replication and storage costs controlled. Objects without locality on their access pattern can be replicated everywhere if desired. We detail this more in Section Replication.

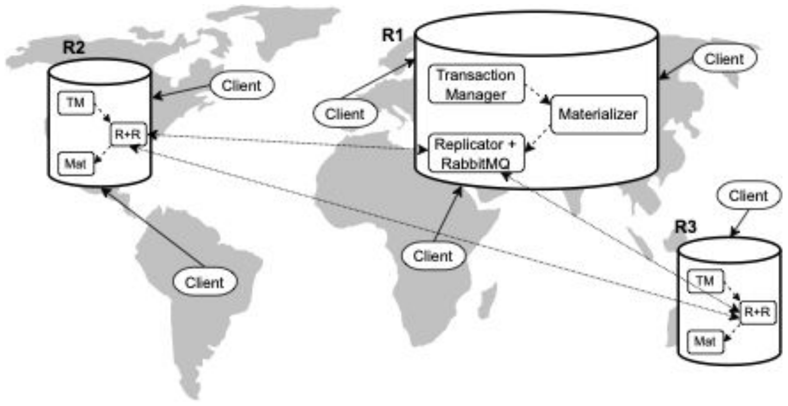


*Figure 2.3.2.1 - PotionDB architecture.*

Clients communicate with the nearest PotionDB location to ensure low latency. A client's transactions are locally executed in the PotionDB's location the client is connected to. Updates are propagated asynchronously to other locations. If a client's transaction accesses objects not locally replicated, other locations with said objects are contacted and involved in the transaction. We note this should be an exceptional case, not the norm.

The internal architecture of each PotionDB server is inspired by Cure [68] and is split into three main components.

First, the Transaction Manager coordinates transaction execution, implementing a transactional protocol. Second, the Materializer stores the objects on their latest version, alongside the necessary data to generate previous versions when necessary. Garbage collection ensures data related with versions that are too old is eventually discarded. Third, the Replicator ensures that committed transactions are replicated asynchronously to other PotionDB instances. It is also responsible for receiving remote transactions and forwarding them to the Transaction Manager for local execution.

### 2.3.2.6 Transaction and replication protocols

This section presents the protocols used to maintain the state of objects and execute transactions with TCC in PotionDB. The transaction processing and replication algorithms are an adaptation of Cure protocols [68] to partial replication.

#### *2.3.2.6.1 Objects*

PotionDB stores CRDTs [69]. CRDTs are replicated objects that are guaranteed to converge. after applying the same set of operations. In particular, PotionDB uses operation based CRDTs, in which the convergence of replicas is guaranteed if operations are causally applied. This is the case in PotionDB, as a valid transaction serialisation must respect the happens-before relation.

Our prototype supports the following CRDTs: last-writer-wins register, for storing opaque values; add-wins set, for keeping a set where adding an element wins over a concurrent removal of that same element; add-wins map, for maintaining a map of values; and counter, for maintaining a number that accepts concurrent increment and decrement operations.

#### *2.3.2.6.2 Sharding*

PotionDB adopts a sharded model, where objects replicated in a location are split into multiple shards. For durability, each shard could be replicated in multiple servers using some replication protocol [72].

In each server, a shard has a dedicated thread, adopting an approach used in other database systems, such as H-store [73]. This avoids using locks when accessing objects, simplifying the implementation and avoiding issues such as lock contention.

#### *2.3.2.6.3 Metadata*

Let $L_1, ..., L_n$ be the set of locations in the system. Each replica $L_j$ keeps a global vector clock $vc_G$, with one entry for each $L_k$ Î $\{L_1, ..., L_n\}$. This clock represents the latest snapshot available in $L_j$, summarising the transactions integrated in the snapshot. Each shard $sh_i$ also keeps a local vector clock $vc_i$. The local vector clock represents the latest snapshot available in $sh_i$, which may be different from $vc_G$. Any shard can access the server's physical clock, *pc*.

A transaction *t* has an associated read vector clock, *t.rc*, that represents the snapshot to be read by the transaction. On commit, a transaction is assigned a commit clock, *t.ct*, consisting in a pair (timestamp, location identifier). A transaction with commit clock $(n,r_i)$ is in snapshot $vc_G$ iff $vc_G[r_i] <= n$.

Each shard maintains a hybrid logical clock (HLC) used to assign timestamps. An HLC uses the physical clock of the computer to generate the next timestamp, unless the physical clock is smaller than a timestamp previously generated/observed. In this case, it returns the maximum previously observed timestamp plus one. This guarantees that timestamps generated are monotonically increasing.

Each shard $sh_i$ also maintains a list of prepared transactions, $prep_i$, and a list of commits on hold, $hold_i$.

#### *2.3.2.6.4 Transaction processing*

A client executes a transaction by interactively contacting a PotionDB server. The transaction execution is coordinated by the Transaction Manager (TM).

When the TM receives a begin operation, it decides which snapshot the transaction will access, i.e., the latest snapshot available at the current location, which is represented by the global vector clock $vc_G$.

When receiving an update operation, the TM asks the Materializer to execute the update in a private copy of the object for the transaction snapshot. When receiving a read operation, the TM asks the Materializer to execute the read operation in the private copy of the object - if no update has been executed before in the object, a shared copy with the version of the transaction snapshot is used. We represent by *t.WS* and *t.RS* the sets of objects updated and read in transaction *t*.

When the TM receives a commit, it needs to assign the commit timestamp to the transaction. For assigning the commit timestamp, the TM runs a two phase protocol with the shards of the objects updated in the transaction.

In the first phase, the TM sends a *prepare* message to all shards in *t.WS*. A shard $sh_i$ replies with a timestamp proposal, *(n,i)*, where *n* is the timestamp generated by the shard's local HLC and *i* is the shard identifier. Additionally, the shard adds the information about the transaction, including the proposed timestamp, to the list of locally prepared transactions, $prep_i$.

The TM collects the replies and sets the commit timestamp of the transaction, t.*ct* = *(mts, j)*, with *mts* the largest received timestamp, and *j* the location identifier. A commit message is sent to all shards in *t.WS* with the commit timestamp.

When receiving a commit message for transaction *t*, a shard proceeds as follows. First, it checks if the commit can be applied, by checking that no prepared transaction or commit on hold has a smaller timestamp. If so, updates from *t* are marked as committed and the shard's local vector clock $vc_i[j]$ is updated with *t.ct*. Otherwise, the commit is queued to be applied later, as other transactions may commit with a lower *ct* than *t.ct*. In either case, the transaction is removed from $prep_i$. If t's proposed value was the lowest, then the Materializer verifies if any commit on hold can now be executed.

The client is informed of the commit as soon as all shards in *t.WS* acknowledge the commit message, even if some shards queued the commit. Additionally, the TM sends information to update the global vector clock to include the timestamp of the committed transaction. Updating the local entry of the global vector clock with the timestamp of committed transactions immediately promotes freshness, as new transactions will always use the latest committed snapshot.

However, this raises a problem, as some shards (not involved in the committed transaction) may still have transactions to commit with smaller timestamps. If a transaction that starts in the most recent snapshot issues a read (or update) operation on object *o* in a shard that has pending transactions with smaller timestamps, the operation needs to block if there is a transaction prepared or on hold that had modified *o*.

##### *2.3.2.6.5 Operations on objects not locally replicated.*

As PotionDB is partially replicated, a transaction may access an object *o* that is not locally replicated.

When an operation is executed in an object *o* that is not locally replicated, the shard that should hold *o* fetches a copy of the object from a remote replica. The version of the object requested is that of the transaction snapshot, t*.rc* ignoring the entry for the current location. It is important to ignore the current location to guarantee a quick reply. Otherwise, as updates are propagated asynchronously, the remote replica would need to wait to receive all transactions from $L_i$ reflected in the transaction snapshot before returning a copy of *o*. After receiving the copy of the object, the shard will apply any updates performed to *o* at $L_i$. Typically, there will be no updates, as $L_i$ does not replicate *o*. After this, the transaction accesses the object as any other locally replicated object. When the transaction commits, the updates to the object are propagated in the context of the transaction.

#### 2.3.2.7 Replication

Buckets are the unit of replication in PotionDB. Each location decides which buckets to replicate. In our e-commerce example, there is a bucket for the customers of each online store. The container *customers* include all customer buckets. The EU customers bucket is replicated in the EU location and in one or more additional locations for fault tolerance.

PotionDB adopts an operation-based replication approach, where transaction updates are propagated to other replicas. To be more precise, in the context of CRDTs, an update executed in a CRDT generates an *effects* operation, and it is this effect update that is propagated and applied in relevant replicas. We start by describing how the replication process guarantees that transaction updates are propagated to all relevant locations. In the end, we discuss the special case of NuCRDTs.

When a transaction commits at a shard, that shard sends its part of the transaction to the replicator. Given how transactions are committed at a shard, it is guaranteed a shard sends the transactions ordered by commit timestamp. If the shard processes no transaction for some time, it notifies the replicator that there are no parts of transactions for the shard until the current timestamp, obtained from the shards' HLC.

The replicator processes the parts of transactions received from shards in timestamp order. For each transaction, the replicator groups the transactions' updates and repartitions the updates, one for each updated bucket. The new transaction parts are then queued for replication, one part for each bucket, being propagated in order to all other locations that replicate the bucket. Note that a location has a logical stream of updates not only for all buckets the location replicates, but also for the buckets that a local transaction has updated an object.

The replicator integrates remote transactions as follows. A replicator subscribes to the logical streams of updates for the buckets it replicates from all other locations. For the logical streams of each location, it processes transaction parts in timestamp order. For a given timestamp, the replicator groups the parts of the transaction *t* and verifies if the causal dependencies of the transaction are satisfied by checking whether *t.rc* <= $vc_G$. If true, the replicator forwards the transaction updates to the local shards for execution. After all shards involved in a transaction *t* end, the global vector clock is updated to include the timestamp of *t*. If false, then some transactions from other locations need to be applied before executing it. So, the replicator continues by processing the logical streams from other locations. Furthermore, the replicator piggybacks to other locations the information about locally executed transactions. For fault-tolerance, a location forwards updates from a failed location to other locations.

#### 2.3.2.8 Status

PotionDB is under development, and we expect to report the evaluation of the prototype in the next deliverable. Additionally, we expect to report the integration of support for recurrent queries and the integration of Potion under the generic API proposed in the context of TarDIS.

### 2.3.3 Integration of Storage Solutions into the TaRDIS Ecosystem (Blockchain, C3, Engage)

#### 2.3.3.1 Overview

Integrating third-party storage solutions into a system poses a range of challenges, emphasising the intricacies of aligning external tools with specific project requirements. Moreover, due to the complexity of distributed storage solutions, having a common interface that could be used by different applications that may wish to rely on different storage solutions proves to be a difficult task, especially in terms of compatibility and interoperability. Achieving seamless compatibility, while offering flexibility to the developer, demands meticulous planning and expertise. The process requires a delicate balance between customization to meet specific requirements and leveraging the capabilities of the external tool.

To address this challenge, we created a set of *adapters*, implemented in the Babel framework, that aims to integrate different storage solutions into the TaRDIS Ecosystem. To enable this, we created a set of abstractions (derived from the APIs presented in Deliverable 3.1 [58]) that offer common interfaces to applications that wish to use any of the different storage solutions provided by TaRDIS and integrate distributed systems protocols into their protocol stack.

The common abstractions aforementioned (materialised as a set of support common classes) support the interaction between well-known classes of distributed protocols by making APIs of these protocols to be generic, allowing implementations to be agnostic to the concrete protocol that materialises a given abstraction. These abstractions are divided into three main categories: *data management*, *dissemination*, and *membership.* These common abstractions are further explained throughout Sections 3.d, 3.e, and 3.f..

With these abstractions in place, we developed a set of *adapters* by leveraging on the interfaces offered in the *data management package*, namely, the *requests* and *replies* abstractions, used to issue operations to the storage layer and to receive their response, respectively. With these abstractions, an application can instantiate any of the client-side adaptors of their choice and communicate seamlessly with the picked storage solution.

As one could expect however, due to the different nature of different storage solutions (i.e., inserting a row in Cassandra presents different semantics from executing a transaction on the HyperLedger Fabric blockchain solution), we offer some flexibility to the developer by allowing different types of operations payloads (materialised as *abstract classes)* when executing an operation. Further details are given in Section 3.11 and in the public repository https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/babel-datareplication-adapters.

### 2.3.3.2 Storage Abstractions

To provide a common interface to different storage solutions, we devised a set of storage APIs which enable an application to interact (i.e., order the execution of operations and obtain their results) with stored data. Following the API style exposed by Babel (that we discuss in further detail on Section 3) the interface to interact with storage abstractions is divided into two different categories: *requests* and *replies,* which allows to execute operations (e.g., create a key space, delete a record, etc.) and receive results from these operations asynchronously, respectively. In more detail, the following abstractions are offered:

**Requests**

- *CreateDataspaceRequest*: Request to create a `dataSpace` in a data management protocol, with a given set of `properties`.
- *CreateKeySpaceRequest*: Request to create a `keySpace`(akin to a table) in a specific `dataSpace` in a data management protocol, with a given set of `properties`.
- *ExecuteRequest*: Request to execute an operation on a specific `dataSpace` and `keySpace`.
  The operation in question is specified through the abstract class `CommonOperation` which can be instantiated with a specific operation type (i.e., `BlockchainOperation`, `PayloadOperation`, etc.).
  Each of these operations follows the design needs of the solution storage being used, namely, the parameters to execute an operation. Additional operation types can be added to accommodate new protocols integrated into the TaRDIS toolbox.
- *DeleteDataspaceRequest*: Request to delete a `dataSpace`.
- *DeteKeySpace*: Request to delete a `keySpace` in a `dataSpace`.

**Replies**

- *CreateReply*: Generated in response to a `CreateDataspaceRequest` or `CreateDataspaceRequest`, with the status of the operation.
- *ExecuteReply*: Generated in response to a `ExecuteRequest`, with the status of the operation and the response data (e.g., a `payload`).
- *DeleteReply*: Generated in response to a `ExecuteRequest,DeleteKeySpace or DeleteDataSpace`, with the status of the operation.
- *NotSupportedReply*: Generated in response to a request for an operation that is not implemented in the underneath data management protocol.

### 2.3.3.3 Current Integrations

At the moment, we provide the adapters for the following storage solutions:

- **Arboreal:**
  This adapter implements Arboreal client-side.
  Arboreal is a generic solution for data management in the edge, that provides causal+ consistency while scaling for a large number of edge nodes (presented in detail in this report on Section 2.c.i).

- **Hyperledger Fabric (Blockchain):**
  This adapter implements Hyperledger Fabric client-gateway, the component in charge of invoking transactions on smart contracts deployed in the fabric blockchain network.

Hyperledger Fabric is a platform for distributed ledger solutions underpinned by a modular architecture delivering high degrees of confidentiality, resiliency, flexibility, and scalability.

- **C3:**
  This adapter implements C3 client-side. C3 [99] is designed to extend existing storage systems by integrating the designed replication schema to enforce causal+ consistency. At the moment this adapter implements C3 integration with Cassandra.

- **Cassandra:**
  This adapter implements Cassandra client-side. Cassandra is a highly performant distributed database, providing high availability and proven fault-tolerance.

- **Engage:**
  This adapter implements Engage client-side. Engage [80] is a storage system that offers efficient support for session guarantees in a partially replicated edge setting.

#### 2.3.3.4 Work In Progress

The adapters shown are currently in development. We expect to add more functionalities to each one (according to the specification on the underlying protocol) as well as provide additional documentation, testing, and examples while following the APIs described above. Additionally, we plan to integrate more data storage solutions into the presented model (e.g., PotionDB that was also presented in detail in this report).

### 2.3.4 Integrating the Actyx middleware: data management for event streams

The Actyx middleware (discussed already in Section 2.3.2) offers not only communication services: by providing durable and reliably replicated event streams it also is used and useful as a data management system. Its focus on recording events in an immutable fashion—using append-only logs—makes it ideal for process auditing, process mining, business intelligence, etc. To this end the included Actyx Query Language (AQL) allows the precise selection of data as well as its transformation, filtering, aggregation, and enrichment using sub-queries. These facilities operate on a more fine-grained level than the data management tasks described for the basic TaRDIS abstractions described in the sections above; it would not be practical to store events that carry just some hundred bytes of information (in many cases less!) directly in a decentralised key–value store, the overhead for managing this storage would be forbiddingly high.

#### 2.3.4.1 Overview of data management within the Actyx middleware

Events in Actyx are stored in log slices, where each Actyx node is the sole editor for a set of such slices. A log slice is a sequence of events ordered by their insertion, meaning that each new event is inserted exactly at the end of the log, becoming the successor of the previously latest event. Which slice an event is added to is determined by the tags it carries. This has been used by Actyx internally e.g. to write metrics events into a slice that is separate from application events.

Actyx uses IPFS [103] to store event data within the swarm, which is a method that identifies and localises data using the cryptographic hash of the data's binary representation. This way, no names need to be managed and agreed upon, storing some binary block of data yields the same hash (called *Content Identifier* or *CID*) regardless of which node is performing the store operation. Finding data by CID requires communication with the rest of the swarm unless the data are already found locally, which means that applying this method to single events is very expensive. Therefore, Actyx created the *banyan* library [106] that partitions the log slice into runs of events and stores each one of these as an IPFS block. Not only does this lead to using fewer CIDs (which leads to less management overhead), it also allows sequences of events to be compressed together before being encrypted, yielding a much better compression ratio than treating each event separately. For illustration, we found that grouping events into blocks such that the compressed size is roughly 100kB leads to a reduction in size by about a factor of 40—this is due to the fact that events from the same source tend to share common syntax and contained data (like IDs or property names) which can then be exploited by the zstd compression algorithm.

Finding events within the compressed data blocks requires indexing information, which is stored in the banyan tree data structure one level above the leaves. The index contains all directly queryable aspects of each event, which includes timestamps (physical & logical) as well as the attached tags and the ID of the application that emitted the event. 32 blocks can be summarised by one branch node, after which the branch nodes are themselves summarised by higher-level branch nodes. The single top-most branch node is referenced from a dedicated root node that contains the encryption configuration for this tree, allowing a log slice to be made readable only to a subset of the swarm nodes—although this is not yet a feature application programmers can use.

#### 2.3.4.2 Adaptations required for Actyx integration into TaRDIS

Some of the swarms targeted by TaRDIS use cases are intended to keep running for extended time periods, leading to the problem that storage requirements scale linearly with the time a swarm has been active. On the other hand, not all details of what some swarm participant did are relevant for ongoing operations many months later—in some cases short-lived workflows are only relevant on the current day or within the current hour. While it remains beneficial to keep some replicas of the full event streams (e.g. in the cloud) for auditing, process mining, and business intelligence, it is also required to free up storage space on the edge devices as soon as possible. To this end, the capability of forgetting some prefix of a log slice needs to be added to Actyx—the event log structure stays immutable, but the actual data of some initial portion of the log may be forgotten. This feature is called *ephemeral event streams*. A directly related feature is the configurable routing of events into log slices because forgetfulness can only be configured at the granularity of a log slice. Pre-TaRDIS Actyx had a fixed set of rules basically separating internal administrative events from application events; this needs to be opened so that the administrator of each Actyx node can decide which events to keep for how long. This declaration will be based on timestamps and event tags.

The above can be done independently of new TaRDIS tools. Once such tools become available, they will also be used to implement a smarter version of the banyan storage format, analogous to using TaRDIS communication primitives instead of the existing libp2p approach. This implies the introduction of TaRDIS interfaces within the Actyx middleware to decouple the core logic of offering fine-grained event streams based on coarse-grained block storage in the swarm from the currently used IPFS implementation. Once this is done, TaRDIS data management can be introduced seamlessly, including intelligent partial replication with dynamic reconfiguration.

#### 2.3.4.3 Current state and future work

So far, we have invested the effort in Actyx to release the ephemeral event streams and configurable event routing features. Like for the communication aspect, we can only start introducing the TaRDIS interfaces once those have been sufficiently stabilised and initial implementations are available for evaluation.

## 2.4 Decentralised Monitoring and Reconfiguration (Task 6.3)

The overall goal of this task is to address the fundamental challenges to create the support for autonomic[6] swarm systems. This requires developing solutions for monitoring and reconfiguration within the TaRDIS toolbox, thus providing support for running swarm applications and the system in highly dynamic environments, where runtime adaptation is relevant, but cannot be executed (at least exclusively) by human operators in an effective way.

The task is being carried out in three parts by designing and developing:

1. decentralised solutions for acquiring telemetry information from components of the deployed system;
2. mechanisms to aggregate and propagate telemetry information to continuously train and enhance ML models via provided APIs, specifically designed for this purpose;
3. solutions for supporting reconfiguration of application components at runtime (e.g., parameters, communication patterns, add/remove components).

In addition to these goals, this task will also focus on the integration with Task 6.1 and Task 6.2 within the same working package, thus allowing seamless integration within the toolbox itself. With Task 6.1, the integration will be carried out in a way to propagate configuration to the nodes and gossip the data to other nodes in the cluster. Integration with the Task 6.2 will be carried out to propagate configuration to the nodes, and store specific configuration elements to the decentralised database, for system and/or application needs.

All developed elements will be carried out using open-source tools, with emphasis on the cloud-edge collaboration.

### 2.4.1 Distributed Management of Configuration based on Namespaces

Distributed systems are usually designed to be able to run multiple applications at the same time (i.e. multi-tenancy) [6]. Such complex systems usually require dynamic configuration based on numerous factors. Dynamic configuration is essential for both effectively managing this large-scale complex shared infrastructure, but also for managing components of complex distributed applications, such as swarm applications, to ensure that they operate correctly and within acceptable performance despite external events.

To be able to provide such functionality, platforms add various isolation mechanisms for available resources, to ensure that one application does not monopolise resources, and starve other participants, a problem widely known as noisy neighbours [7]. Running applications in isolated environments (e.g. containers, virtual machines, unikernels, etc.) goes a long way to ensure that during the application running time, resource spread will be fair and as described by clients.

On the other hand, we must ensure that the system does not even come to the point that these applications create problems on start, but also to allow collision-free naming that is usually overlooked. The system should provide logical isolation of both resources and naming on various objects that exist in the system.

[6] Autonomic is a term popularised in a seminal work of IBM [107], that captures that notion of a system that can self-manage and self-optimise.

Since namespaces allow the system to isolate applications and their artefacts, this could be useful for other parts, for example, configuration and telemetry acquisition. With different strategies, users can configure applications running inside logical isolation, or namespace, differently than applications in other namespaces. This allows the system to be able to run, configure, and manage virtual clusters/swarms inside physical clusters/swarms based on different criteria. At the same time, it allows the system to extract telemetry information based on usage in every single namespace and allows different data transformations on smaller chunks of data when needed.

The last benefit that namespaces bring is the ability for users to test their applications in a contained real-world environment. With this, users are allowed to do experimentation with their applications on small, isolated spaces, and new users can practise, and get to know the system.

Based on this we have developed a mechanism, which we integrated with docker containerization environments, where we can manage components of different distributed applications based on hierarchical namespaces. To do this we allow different components of a system to be both associated to a namespace and be characterised (by the developer) by a set of labels, which can be used to determine whose components of the application will be affected by reconfiguration/management actions.

Labels represent an arbitrarily long array of key-value pairs, usually attached to many application components (or more generally, objects) in the system. This idea is usually used to provide easy and rich query mechanisms, in complex systems operating on many nodes, running multiple applications on these nodes with many other abstractions (e.g. configurations, secrets, actions, etc.) coexisting on these nodes at the same time. Kubernetes [1] uses these ideas to specifically identify attributes of objects that are meaningful and relevant to users but do not directly imply semantics to the core system.

The main goal of labels is relatively simple, their existence allows users to simply map their own organisational structures onto abstract objects provided by the system, in a loosely coupled fashion [1, 2]. Clients are not required to store these mappings. Labels can be added to object creation [2], but can also change during the object lifecycle, at any point in time.

They shine in complex systems, because of their loosely coupled nature [3], allowing both system and users to easier track, query, and manage multiple objects, sometimes even at the same time [4]. The users submit their tasks to the system, and the system will do a selector query to find the right infrastructure objects, based on provided labels, and tie everything together in one working unit, a technique known as infrastructure as software [5].

With labels, we can point a system to propagate a message directly to the node, or group of nodes without any direct knowledge of where that node is. This could be applied furthermore to any other object running on the node, or group of nodes. Message propagation could be implemented in event-based fashion using open-sourced industry standard tools such as Apache Kafka, NATS etc.

One interesting effect from this approach is that extension is relatively easy. The user needs to hint to the system where to look for infrastructure objects, and based on the provided labels, the rest of the system can run the same type of operations – query provided labels. Hinting information to the system is relatively easy, since every infrastructure object has a unique name, representing that exact resource, the system can get a proper pool of resources and do the rest of the query on existing labels and provided selector.

### 2.4.2 Telemetry Acquisition for Decentralised Systems

Currently developers of swarm and decentralised applications have no standard and easy way to collect runtime telemetry information about the operation of these applications, meaning developers must log and then parse all relevant information that could give them an insight into the correctness or effectiveness of their developed solution, which is impractical. Alternatively, they could integrate existing metrics collection systems which is time consuming and may not provide optimal results as the existing solutions and architecture (e.g., Nagios, Prometheus) are not tailored for the particularities of decentralised systems, meaning they may not suit the needs of developers or may cause performance issues when handling metrics from a large number of application nodes.

To overcome this lack of support, we have conducted in this activity the development of new mechanisms with the goal of allowing developers to instrument their applications to monitor application/protocol specific runtime indicators, general hardware, and arbitrary metrics, while minimising the interference of these mechanisms with the code of distributed protocols and applications.

To achieve this, our proposal must, not only, provide support for the most common metric types, such as, counters, gauges, and histograms, but also, offer sets of ready to use metrics depending on the type of protocol or application being developed. For example, if the developer is creating a variation of the Paxos [84] agreement protocol, they may be interested in collecting metrics that are relevant for general agreement protocols, such as the average time or number of messages needed to reach agreement, and as such we want them to be able to have these metrics easily accessible to use.

The collected metrics must then be exported, so they can be used to observe the behaviour of the system. These can be exported using a multitude of methods, such as exposing it through HTTP endpoints, exporting it to a log file, publishing them to a Message Queue to be processed, etc. In the future, these metrics might be fed to a distributed subsystem that analyses them and based on that, and potential machine learning derived models of a concrete application, makes automatic management decisions.

Since exporting can take place using varying methods, the system must support different formats to structure the metrics to be exported. These should include widely used formats, such as Prometheus Text Format and OpenTelemetry Format, enabling developers to use this system with existing monitoring tools which they may already be using, but also allow developers to specify their own formatting.

This solution will also include an application that serves as monitor to receive and display the exported metrics, which will initially be centralised. However, this is incompatible with decentralised applications that feature large numbers of nodes working together. As such, another key feature of this solution will be network aggregation, meaning nodes exporting metrics will also act as receivers of metrics from other nodes, aggregating the received information with their own, propagating aggregated metrics information along the network, allowing for multiple points of monitoring of the system.

#### 2.4.2.1 Architecture

In this section we present the architecture of our proposed solution for the problem of collecting and exporting metrics in decentralised systems. The solution is based on the following design principles:

- The system must be able to collect metrics from different protocols, as decentralised systems are composed of different protocols that interact with each other;

- The system must be able to export the collected metrics in different formats, as different systems may require the metrics to be exported in different formats;
- The system must enable developers to create and use new exporters, so as to adapt to many types of monitoring systems;
- The system must be able to be reconfigured without the need to recompile the code, as the number of nodes in a decentralised system may be large;
- The system must be able to be used to collect insight about different types of decentralised systems, as the metrics collection and exporting system should be as general as possible;
- The system architecture must allow for the system to be replicated in many programming languages, as decentralised systems are implemented in different programming languages.

Taking into account these principles, we propose a solution based on the following components, which are described in more detail in the following sections:

- The Metrics Manager, the core of the solution, responsible for managing and mediating interaction with the other components;
- A set of types of metrics, such as Counters, Gauges and Histograms, which are used to collect different types of data;
- A set of Metric Registries, which store, for each protocol, the metrics to be collected and exported;
- A set of Exporters, responsible for exporting samples of the metrics.

**Metrics Manager:**

As previously mentioned, the Metrics Manager component is responsible for managing and mediating interaction with the other components that comprise the metrics collection and exporting system. As such, all user interactions with the system are performed through this component. There must only be one instance of this component per process.

This component is responsible for the following tasks:

- Registering metrics in the Metric Registry with the given id, as requested by a user, creating the registry if one with the given id wasn't yet created;
- Registering exporters, either when the corresponding method is called by the user or using configuration files;
- Instantiating the registered Exporters, by assigning a thread to each one;
- Responding to requests for metric samples by Exporters by asking Metric Samples to be generated by the corresponding Metric Registries.

The choice to allow for configuring the exporters to be registered and initialised using both method calls, and configuration is explained by the fact that using configuration files allows for a more flexible and dynamic system, as it allows for the system to be reconfigured without the need to recompile the code, which is useful in systems where the number of nodes is large.

On the other hand, using method calls allows for a more controlled and predictable system, as the method signature and the compiler will enforce the correct parameters are being used to initialise the exporters.

**Metric Registries:**

The need for placing metrics within what is known in this system as Metric Registries stems from the fact that decentralised systems require interactions between different protocols to achieve their desired purpose, such as Membership Protocols, which keep track of which nodes are part of the system, and Agreement Protocols, which allow nodes to agree on the order of operations to execute. -> (maybe i need citations for what are membership and agreement)

Since each protocol has its own set of metrics, which we may be interested in collecting at different times or using different methods, each instance of a Metric Registry keeps all metrics for a specific protocol we want to monitor.

When an Exporter requests a sample of the metrics of a certain protocol, the Metrics Manager asks the corresponding Metric Registry to generate a sample of the metrics, which is then returned to the Exporter.

**EpochSample:**

The EpochSample is a data structure that contains a set of MetricSample collected from a single protocol, and an associated timestamp, such timestamp is not expressed in seconds but in epochs, an arbitrary time measurement that is increased each time the metrics of a certain protocol is collected. Notice that epochs do not have to be globally synchronized across all nodes in the swarm.

This choice is justified by the fact that using physical time in systems where we have large numbers of nodes to correlate whether the metrics were collected in that same timeframe is not possible, thus we use the notion of epoch.

**Metrics and Metric Samples:**

Metrics currently fall into 3 categories:

- Counters, a value that only goes up;
- Gauges, a value that can go either up or down;
- Histograms, used to divide observations in customizable intervals;
- Simple registers that can have a non-numerical value (potentially out of a list).

When each metric is collected a MetricSample is generated, which encompasses the current value of the metric, or an aggregation of those values if requested, and a timestamp.

**Exporter:**

The Exporter component is an abstract component that provides the base for the implementation of different types of exporters. It offers ways to collect metrics samples from the Metrics Manager for one or more protocols and to load configurations related to the exporter using configuration files.

Since each exporter must implement the method that is run within the thread assigned to each exporter by the Metrics Manager, they are fully in control of how frequently the samples are generated and how those samples are exported.

This allows for the developers to implement the exporters in a way that is best suited for the system they are working on, as they can choose the best way to export the metrics.

The system includes a set of exporters that are already implemented, such as the TimedLogExporter, which exports the metrics to a log file at regular intervals, and the

HTTPExporterPrometheus, which exposes the metrics in a format that can be scraped by a Prometheus server.

## 2.5 Plan and Future Activities

Up to this point we have reported on the main completed activities in the context of the different tasks of WP6. As stated before, in the reported period WP6 has focused on providing fundamental support for the development of swarm applications across the several directions covered by WP6, namely membership and communication abstractions, distributed data management, monitoring and autonomic control. Across these directions we explored new directions and novel solutions, and made available reusable components of previous existing solutions to simplify their integration in the development of swarm applications within the TaRDIS ecosystem.

In the remainder of the project, WP6 is going to continue to develop and integrate technology within these main vectors, by addressing challenges such as security, and the integration of runtime support technology across the other components of the development cycle, such as verification of correctness. The focus of each of the WP6 tasks in the next cycle of development of TaRDIS is discussed in the following.

### 2.5.1 Task 6.1

Task 6.1 will focus on addressing aspects related with security on membership and communication abstractions. This will entail defining mechanisms for dealing with challenges such as identity management at the membership layer, which will require any element of a swarm to prove that it is part of the swarm before being accepted at the membership layer, and hence before having access to any swarm service or functionality of the application. Security questions will also be addressed in the context of communication abstractions, due to the need to ensure integrity and privacy over the information exchanged between different participants of the swarm. These efforts are planned to have an impact on the evolution of the Babel framework, as it will force us to rethink the operation of the framework as to expose adequate abstractions to simplify the development of distributed protocols and applications that require security mechanisms. An interesting application of these efforts will be to devise novel decentralised membership services and communication mechanisms based on publish-subscribe that can provide privacy to support open markets, such as the one in the context of the EDP use case.

An important aspect that was not yet tackled by WP6 is the definition of membership abstractions based on partial information of the system that can consider application specific proximity metrics. Such membership abstractions are important for several reasons, one of them being efficiency, since if we operate on top of a membership service that provides geographical proximity, for instance, most interactions on the swarm will be more reliable and incur in a lower operational cost. Similarly, if we consider a proximity metric based on administrative domains, interactions among nodes in the same administrative domain might benefit from the fact that some information can be shared with lower concerns for privacy or ownership. As detailed on Deliverable 2.2 [96], application-specific proximity bias on membership services is relevant in different swarm application scenarios.

Another direction that will be explored in the context of Task 6.1 in the future is to define novel solutions for decentralised publish-subscribe, data streaming, and multicast. This is a necessity for coming up with mechanisms to aggregate and propagate monitoring information that will govern the self-adaptation of complex swarm systems (in task 6.3). Within this content, we are also interested in identifying mechanisms that could provide some form of privacy or anonymity for participants using point-to-multipoint communication abstractions such as publish-subscribe or multicast.

### 2.5.2 Task 6.2

The solutions developed so far in the context of Task 6.2, and reported here, have been restricted to distributed storage solutions that operate in dedicated infrastructures, that while being useful to several swarm applications, limits its applicability to settings where an infrastructure is permanently in place and available. In the next development cycle of TaRDIS, Task 6.2 will focus on developing decentralised storage management solutions, that are achieved by the combined efforts of other components of a swarm application, not requiring a specialised dedicated infrastructure. Evidently, achieving this will require research, since such a system must face clear challenges related with the durability and availability of shared data across different swarm components. Moreover, such a solution will require proactive replication of data to deal with these challenges, which can be challenging in settings where the number of available resources across different swarm components are limited (in contrast with the virtually unlimited resources of cloud infrastructures).

Similarly to Task 6.1, Task 6.2 in the next cycle of development of TaRDIS will address issues related with security in the context of distributed storage management systems. In this context it is relevant to consider what are the implications for a swarm system when a participant in a distributed storage system can exhibit a byzantine behaviour. Typical cloud-based, and even edge-based, solutions assume that components of the distributed storage systems operate on devices that can be somewhat trusted, whereas the operator can look at the data, but will not delete or modify that data stored arbitrarily (this is aligned with the curious but honest fault model). To address this challenge, we plan to consider the implications of byzantine nodes (at the replica level) on consistency models and replication protocols, allowing us to devise novel decentralised solutions that provide clear guarantees in the context of swarm systems.

Finally, also in alignment with efforts to be pursued in Task 6.1 and described above, we plan to explore a novel location-centric data model and replication strategy, where we associate to data objects of a swarm application a location property that becomes a primary descriptor of the data elements (akin to the timestamp in time series databases) which can then be used to govern processes such as data placement, replication protocols, consistency guarantees offered to different entities manipulating that data, in a way that is automatically defined based on the location associated to data elements and the location of the entity accessing the data. Evidently, location here can be a geographical location, which makes sense on global scale systems, but it can also be a logical location reference, such as an administrative domain.

### 2.5.3 Task 6.3

In the next development cycle of TaRDIS, Task 6.3 will focus its efforts on three complementary and related tasks. In relation to the acquisition of telemetry at runtime from swarm systems, this task will explore scalable mechanisms to make the data available on locations where it can be used to, on one hand, train machine learning models that can guide the management of systems under evolving conditions; and on the other hand, to locations that effectively can plan reconfiguration actions over the system and coordinate those actions across small segments of the system (as to avoid expensive, and many times unfeasible, global coordination mechanisms).

A second direction to be explored in the context of Task 6.2 is to develop decentralised approaches to manage swarm systems components. The solution discussed in this document, while organising the system components on hierarchical namespaces, still relies on a centralised control architecture. In the next development cycle of TaRDIS, we plan to distribute the control component also following the hierarchical organisation of these namespaces, as to minimise the need for coordination, and to allow the system to be able to evolve, even in scenarios where some components are not reachable (for instance, due to a transient network partition). Such an approach will also limit the scope where telemetry information is required to guide the self-management process.

Finally, Task 6.3 will make efforts to integrate with decentralised machine learning mechanisms, such as to enable the training of models with information that produces different logical areas of the swarm, without needing to concentrate that information at a single location, which could not be feasible due to the large amount of information that might be required for these processes.

# 3 SOFTWARE

## 3.1 OVERVIEW

In addition to conduct the research and development of novel solutions to address decentralised membership and communication primitives (Task 6.1), novel distributed data management solutions (Task 6.2), and new techniques to acquire telemetry on the execution of swarm systems to coordinate its runtime adaptation moving toward self-managed swarm systems (Task 6.3), WP6 also is responsible for producing and making available software pieces. These software pieces serve as demonstrators of the advances produced by the project in these directions, and that can serve as reference implementations for these solutions, which can assist the community in developing their own implementations of these systems specially tailored for their execution environments. Some of these demonstrators will be integrated into the TaRDIS toolbox, as discussed on Deliverable 2.2 regarding requirements [96] and Deliverable 3.1 regarding envisioned APIs [58].

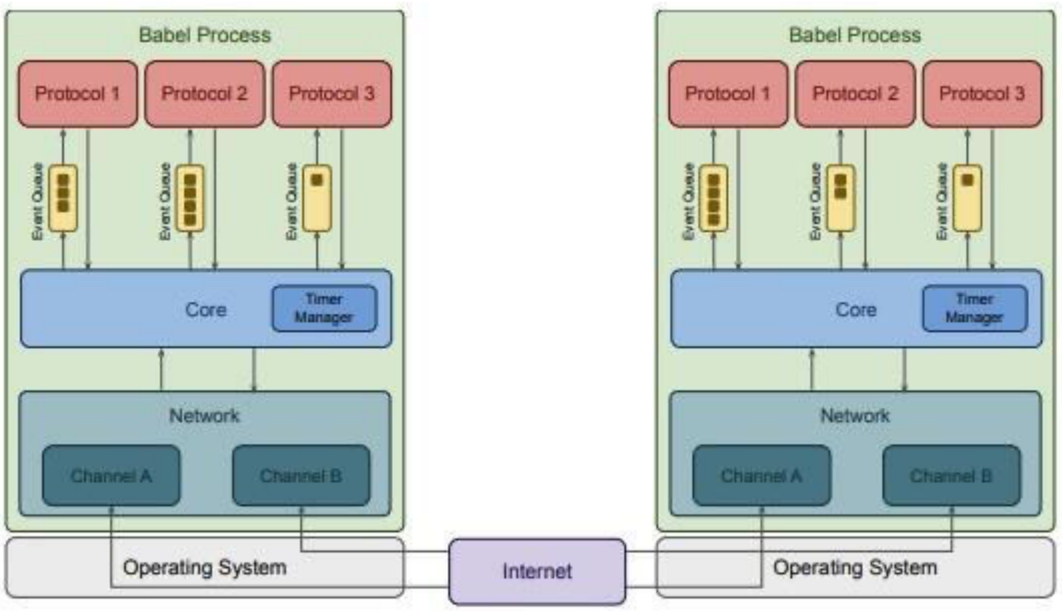
This section provides pointers and small descriptions of the software artefacts that are part of this deliverable, point to the public repositories of where this software can be accessed, and provide - when applicable - descriptions of how to use it. We start by discussing some software pieces that have supported our development of these artefacts (Sections 3.3 - 3.12). In the following sections we discuss the software artefacts of the results reported previously in Section 2, and additional development that we did to support the integration of these artefacts.

## 3.2 PRELIMINARIES: BABEL FRAMEWORK

As discussed on other TaRDIS deliverables, namely Deliverable 3.1 [58], the TaRDIS consortium is adopting an event driven programming model. This is highly suitable for the development of novel decentralised protocols within the context of WP6, since as exemplified in this document, the specification of these protocols is many times in the literature done using an event-drive model (akin to actors' models), which allow to discuss fundamental ideas and approaches in a way that is (mostly) independent of the programming language.

However, we had to make decisions on which programming language (and environment) to use for building demonstrators and reference implementations to some of these solutions, particularly in a way that would allow these building blocks to be easily reused and composed to develop different swarm applications and systems. As mentioned previously in this document, we have decided to implement some of these components using the Babel Framework [59], a java framework previously developed by members of the NOVA team, whose goal is to simplify the development (and execution) of distributed systems, and in particular decentralised system, through the composition of protocols. For self-containment, we now provide a brief description of the Babel framework, that is required to understand some of the development on other WP6 artefacts.

Babel is a framework that aims to simplify the development of distributed protocols within a process that executes in real hardware. A process can execute any number of (different) protocols that communicate with each other or/and protocols in different processes. Babel simplifies the development by enabling the developer to focus on the logic of the protocol, without having to deal with low level complexities associated with typical distributed systems implementations. These complexities include interactions among (local) protocols, handling message passing and communication aspects, handling timers, and concurrency-control aspects within, and across, protocols (while enabling different protocols within a process to progress independently). Notably, Babel hides communication complexities behind abstractions called channels that can be extended/modified by the developer, with Babel already offering several alternatives that capture different capabilities (e.g., P2P, Client/Server, ϕ-accrual Failure Detector). Babel is implemented in Java, taking advantage of its inheritance mechanisms, such that developers extend abstract classes provided by the framework to develop their own protocols and solutions. The strong typing provided by Java allows the framework to easily enforce expected behaviour, while at the same time offering enough flexibility for the developer to implement any type of distributed protocol or system.



The figure above illustrates the architecture of Babel. In the example, there are two processes executing Babel, each process being composed of three (distributed) protocols and two network channels for inter-process communication. Naturally, any distributed system operating in the real world will be composed of more than two processes. The Babel framework is composed of three main components, which we now detail.

### 3.2.1 Protocols

Protocols are implemented by developers (i.e., the users of the Babel framework), and encode all the behaviour of the distributed system being designed. Each protocol is modelled as a state machine (or an actor as in actor-based systems such as Scala) whose state evolves by the reception and processing of (external) events. For this purpose, each protocol contains an event queue from which events are retrieved. In the context of Babel, these events can be Timers, Channel Notifications from the network layer, Network Messages (most of the times originated from another process in the system), or Intra-process events used by protocols to interact among each other within the same process.

Each protocol is exclusively assigned a dedicated thread, which handles received events in a serial fashion by executing their respective callbacks. In a single Babel process, any number of protocols may be executing simultaneously, allowing multiple protocols to cooperate (i.e., multi-threaded execution), while shielding developers from concurrency issues, as all communication between protocols is conducted through the asynchronous exchange of events with no form of memory sharing between different protocols, even within the same babel process.

From the developer's point-of-view, a protocol is responsible for defining the callbacks used to process the different types of events in its queue (i.e., event handlers). The developer registers the callback for each type of event and implements its logic, while the execution engine of Babel handles the events by invoking their appropriate callbacks. While relatively simple, the event-oriented model provided by Babel allows the implementation of complex distributed protocols by allowing the developer to focus almost exclusively on the actual logic of the protocol, with minimal effort on setting up all the additional operational aspects, including handling concurrency challenges, complex asynchronous interactions, among others.

### 3.2.2 Core

The Babel core is the central component which coordinates the execution of all protocols within the scope of a process.

As illustrated in the figure on the previous page, every interaction in Babel is mediated by the babel core component, as it is this component's responsibility to deliver events to each protocol's event queue. Whenever a protocol needs to communicate/interact with another protocol, it is the core that processes and delivers events exchanged between them. When a message is directed to a protocol in another process, the core component delivers it to the network channel used by the protocol, which then sends the message to the target network address. That message is then handled by the babel core of the receiving process that ensures its delivery to the correct protocol.

Besides mediating interaction between protocols (both inter and intra process), the core also keeps track of timers configured by protocols and delivers an event to a protocol whenever a timer defined by that protocol is triggered. This avoids complex (and potentially operative system dependent) tasks to schedule the execution of actions by time.

### 3.2.3 Network

Babel employs an abstraction for networking named channels. Channels abstract all the complexity of dealing with networking, and each one provides different behaviours and guarantees. Protocols interact with channels using simple primitives (openConnection, sendMessage, closeConnection), and receive events from channels whenever something relevant happens. These events are channel-specific and are handled by protocols just like any other type of event (i.e., by registering a callback for each relevant channel event).

For instance, the framework provides a simple TCPChannel which allows protocols to establish and accept TCP connections to/from other processes. This channel generates events whenever an outgoing connection is established, fails to be established, or is disconnected, and whenever an incoming connection is established or disconnected. Other examples of provided channels include a channel with explicit and automatic acknowledgement of messages, a channel that creates one connection for each protocol running in different processes, and a ServerChannel that does not establish connections, only accepts them, and its corresponding counterpart, the ClientChannel which behaves in a symmetrical fashion. We also provide a TCP-based channel that implements the ϕ-accrual failure detector [97], which notifies protocols that registered a callback whenever another process is suspected.

The Babel framework also allows developers to design their own channels if they need to enforce some specific behaviour or guarantee at the network level for a protocol to function correctly. Network channels are implemented using Netty [98], which is a popular Java networking framework. However, the typical developer of Babel does not have to interact with Netty directly.

A protocol can use any number of channels, and a channel can be shared by more than one protocol. In the example shown in the previous figure, two channels were instantiated by Babel. Channels within a Babel process is instantiated on demand by the Babel core when protocols are instantiated. Upon protocol instantiation, protocols define the channels they will be using, instructing the Babel core to prepare and start the necessary network channels. Network channels have their own execution threads allocated to them, since they are responsible for serialise and deserialise events that are sent to (respectively, received from) the network, among other tasks.

### 3.2.4 API and Babel Events

Babel is provided as a Java library. Protocols in Babel are developed by extending an abstract class - GenericProtocol. This class contains all the required methods to generate events and register the callbacks to process received events. Each protocol is identified by a unique identifier, used to allow other protocols to interact with it. There is also a special Init event that protocols must implement, which is usually employed to define a starting point for the operation of the protocol (e.g., communicate with some contact node already in the system or set up a timer event).

The API can be divided in three categories: timers, inter-protocol communication (within the same process), and networking, which we discuss briefly below. These define the types of events that a protocol can handle.

**Timers**

Timers are essential to capture common behaviours of distributed protocols. They allow the execution of periodic actions (e.g., periodically exchange information with a peer), or to conduct some action a single time in the future (e.g., define a timeout).

In Babel, using timers can be achieved as follows. First, the developer needs to create a Java class that represents a timer, with a unique type identifier and extending the generic ProtoTimer class. Additionally, a timer might have any number of fields or logic as the developer needs (i.e., Timers in babel are Java classes that can have state and methods to manipulate that state). To use a timer in a protocol, a callback method must be defined to be executed once the timer expires (a process that is controlled by the Babel core). This method must receive as parameters the timer object and its instance id, which is generated when a time is set up by a protocol. This callback is registered by calling the method registerTimerHandler, which takes as arguments the unique type identifier of the timer, and the callback function itself.

After registering the handler, any number of single-time or periodic timers can be set up by a protocol using the methods setupTimer or setupPeriodicTimer, respectively. These methods take as parameters an instance of the timer class, and the delay to trigger the timer (in milliseconds). The periodic timer also requires a third parameter: the periodicity after the first triggering of the time, in which copies of that Timer will be triggered again. Cancelling timers is also possible – for this, we simply call the method cancelTimer with the identifier of a previously setup timer as parameter.

**Inter-Protocol Communication**

The Babel framework supports multiple protocols executing concurrently in the same process. As such, we offer mechanisms for these protocols to interact with each other, allowing them to cooperate or delegate responsibilities among them.

To support this, Babel provides two types of communication primitives: one-to-one requests/replies and one-to-many notifications. Similarly to timers, requests, replies, and notifications need to extend a generic class (ProtoReply, ProtoRequest, and ProtoNotification respectively) and have unique type identifiers. Any of these events can have arbitrary state and/or logic. Again, similarly to timers, we need to register a callback for each type of event processed by a protocol. These callback functions all take the same parameters: the object that was sent and the identifier of the protocol who sent it. To send requests and replies, the methods sendRequest and sendReply are used. These methods take as parameters the Request or Reply to be sent and (numerical identifier of) the destination protocol.

Notifications, however, are different since they are triggered by a protocol without being aware of which other protocols (if any) have registered the interest in processing these. Due to this, the method triggerNotification does not take a destination protocol, instead, every protocol that subscribed to that type of notification receives a copy of this event. This is handled internally by the Babel core.

**Networking**

Naturally, as a framework for distributed protocols, Babel also provides abstractions to deal with networking (including management of connections). For this, we provide different network channels with different capabilities. The interaction of protocols with channels is (mostly) similar across different channels.

To use a channel, we start by setting up the properties for that channel. Considering the example of the TCPChannel, the required properties are the binding address and port for the listen socket; other channels can consider different properties (e.g., a server channel that takes as property the maximum number of simultaneous client connections).

A channel is effectively created by calling the method createChannel, passing the name of the channel and the properties object. This method returns an identifier representing the created channel. This identifier is useful if a protocol uses multiple channels simultaneously, to be able to select which channel to use to send a specific message, and to register different callbacks for different channels. Similarly to timers and requests/replies, we also need to create a class for each network message to be sent through a channel. Besides extending a generic class, and having a unique type identifier, the developer must also define a serializer for each message to enable the message to be encoded and decoded into network buffers.

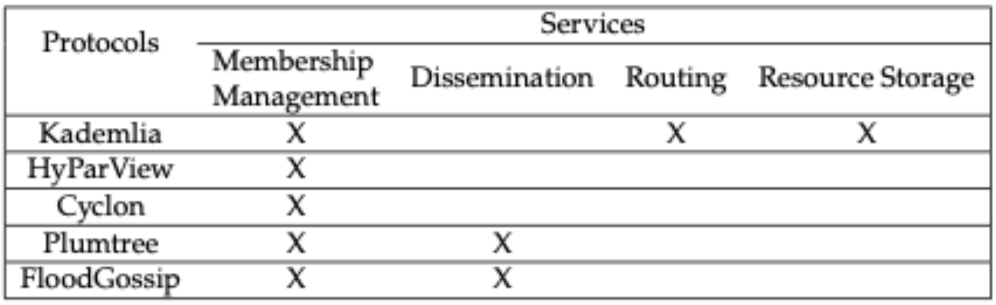
A message can be sent using the sendMessage method, which takes as arguments the message to be sent, the destination address/port, the channel identifier (if more than one channel is being used) and optionally, the destination protocol. An additional parameter representing the connection to use can be passed. The interpretation of this parameter is, however, channel dependent. Finally, each channel is responsible for generating notifications for relevant events that occur in it, for which the protocol can register callbacks.

## 3.3 A Generic API for Decentralised Overlay and Communication Protocols

<u>Repository:</u>
https://codelab.fct.unl.pt/di/research/tardis/wp6/overlayapi

The code containing our proposed implementation of a Generic API for Decentralised Overlay and Communication protocols is available at https://codelab.fct.unl.pt/di/research/tardis/wp6/overlayapi.

All components described in Section 2.b.i.3 are available in our implementation, this includes the Protocol Manager, a set of generic interfaces to interact with each service devised, multiple decentralised protocols, and a set of applications developed in order to demonstrate, test and evaluate the proposed solution. The four services discussed in detail in Section 2.b.i.2 (Membership Management, Dissemination, Routing, and Resource Storage) were also implemented. The table below presents an overview of the protocols available in our



| Protocols | Services | | | |
|---|---|---|---|---|
| | Membership Management | Dissemination | Routing | Resource Storage |
| Kademlia | X | | X | X |
| HyParView | X | | | |
| Cyclon | X | | | |
| Plumtree | X | X | | |
| FloodGossip | X | X | | |

implementation together with the services provided by each one.
Multiple examples of applications that take advantage of the software developed are available at the project repository and are located in the `src/main/java/applications` directory. These applications are the ones described when presenting the experimental validation in Section 2.b.i.4. Applicational examples are divided between automated and interactive ones with some, such as the dissemination and routing applications, having both implementations while others are only provided with one type of implementation, this is the case of the peer-sampling application (only automated) and resource storage application (only interactive).
The applications provided serve not only as a means of demonstrating, testing, and validating the solution implemented but also as a reference to understand how the provided components should be leveraged when implementing new applications based on the solution proposed for interacting with decentralised protocols. This includes how the interaction with the Protocol Manager should be performed, namely regarding its instantiation and the request of new decentralised protocols, and the interaction with the generic interfaces of each service provided, taking advantage of each one of the generic, synchronous and asynchronous, mechanisms made available by our solution.
To configure the operation of the software some configuration files should be considered. First, the `config.properties` file at the root of the repository should be used as the main configuration file, containing protocol-specific configurations as well as system-specific ones. Additionally, each application also contains a dedicated configuration file in the directory where it is located where all application-related configurations are placed.
To run the software a Java 8 installation is required as well as the installation of the Maven tool, required for performing the build and packaging operations. Therefore, before running the software a `mvn compile package` command should be executed, from the root of the project, to build the required jar file. Then, to launch the applications, a testing script is available in `scripts/test-script-local.sh` which, by default, launches 10 instances of the routing application that perform routing requests taking into consideration the configurations defined in each configuration file. After the execution of the routing applications, 10 instances of the dissemination application are also launched by the script. The versions of the applications considered by the script are the automated ones that perform the routing/dissemination operations without any intervention during a predefined amount of time. Each application can also be tested individually by executing the respective code and providing the necessary configurations. As an example, the following commands, executed from the root directory of the project, launch two instances of the automated routing application, one in port 10000 and other in port 10001 (with the first one as contact node). The configurations related

with the execution of the underlying API mechanisms and protocols should be configured in the properties file located at the root of the project. These configurations include the address and port where each protocol instance (represented by its identifier) should run, the configuration of a set of parameters for each available protocol, as well as other system-specific configurations. Moreover, the configurations can also be overwritten through command line arguments, as shown in the example below when considering the port configuration.
The configurations related with the execution of the application, e.g., the execution time, should be defined in the properties file located in the routing application directory.

```
java -cp target/DecentralizedAbstractions.jar
applications.auto.routing.RoutingApp protocol.10000.port=10000

java -cp target/DecentralizedAbstractions.jar
applications.auto.routing.RoutingApp protocol.10000.port=10001
contacts=127.0.0.1:10000
```

Additionally, the remaining project directories contain the implementation of the components required by our solution namely the protocols, available at `src/main/java/protocols`, and the API-related components, i.e., code related with the implementation of generic interfaces and Protocol Manager, available at `src/main/java/api`.

## 3.4 Membership Abstractions Implementations in Babel

Repositories:
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/babel-protocolcommons
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/protocols

The current membership abstractions are implemented in the TaRDIS gitlab https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/protocols

The code in the first repository implements the simplified API for these protocols , that was derived from the work presented previously. This API is provided as a library for the Babel framework (that is compatible with the main distribution of Babel-Core and its variant with support for telemetry acquisition), and can be summarised as:

### 3.4.1 Requests and Replies

● *pt.unl.fct.di.novasys.babel.protocols.membership.requests.GetNeighborsSampleRequest*: Request to get a sample of neighbours (Host format) from the Active view up to a number (provided in the message).
● *pt.unl.fct.di.novasys.babel.protocols.membership.requests.GetNeighborsSampleReply*: Generated in response to the previous request.

### 3.4.2 Notifications

● *pt.unl.fct.di.novasys.babel.protocols.membership.notifications.NeighborUp*: indicated the Host of a local neighbour that became available;
● *pt.unl.fct.di.novasys.babel.protocols.membership.notifications.NeighborDown*: indicates the Host of a local neighbour that is no longer available.

### 3.4.3 Abstractions

- HyParview: An implementation of the HyParView protocol [22], an unstructured overlay network  that is known for being extremely resilient to the concurrent failure of nodes. HyParView operates using two independent partial views managed using different strategies, that ensures that the protocol can - in a fully decentralised fashion - adapt itself to changes in the membership or event to churn scenarios.
- X-BOT: An Implementation of the X-BOT protocol [43] an evolution of the HyParView protocol that allows the overlay network maintained by the protocol to self-adapt to promote links given an application specific criterion captured by a local oracle. This implementation only considers as optimization criteria the latency between nodes (meaning that low latency links are preferred).

### 3.4.4 Usage

To use the aforementioned classes, a user must first create (or use a previously created) Java project with the Babel-core and the Babel-ProtocolCommons dependencies.
With this, to make use of the provided abstractions, while developing its membership protocol (or use a previously developed one) a user must incorporate them with the inter-protocol communication functionality offered by Babel, namely, by providing the request handlers in conformation with the API, and by sending information through *sendReply* and *triggerNotification* methods of Babel.

A few examples of these implementations can be found in babel-protocols.

## 3.5 Communication Abstractions Implementations in Babel

Repository:
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/babel-protocolcommons
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/protocols

The current membership abstractions are implemented in the source code made available at repository babel-protocolcommons under the package *dissemination*.

At the moment of writing this report the following abstractions are provided:

### 3.5.1 Requests and Replies

● *BrodcastRequest*: Request to propagate a broadcast message (in the name of an Host) with the provided `payload` and a specific`TTL`.

### 3.5.2 Notifications

● *BrodcastDelivery*: Indicates that a broadcast message with a `payload` has been received.

### 3.5.3 Abstractions

- Flood Broadcast protocol;
- Eager Gossip Broadcast protocol.

### 3.5.4 Usage

To use the aforementioned classes, a user must first create (or use a previously created) Java project with the Babel-core and the Babel-ProtocolCommons dependencies.

With this, to make use of the provided abstractions, while developing its communication protocol (or use a previously developed one) a user must incorporate them with the inter-protocol communication functionality offered by Babel, namely, by providing the request handlers conforming with the API.

A few examples of these implementations can be found in babel-protocols.

## 3.6 Data Management Abstractions Implementations In Babel

Repository:
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/babel-protocolcommons
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/babel-datareplication-adapters

The current data management abstractions are implemented in the code made available in the public repository babel-protocolcommons under the package *datamanagement*.

At the moment the following abstractions are provided:

### 3.6.1 Requests

● *CreateDataspaceRequest*: Request to create a `dataSpace` in a data management protocol, with a given set of `properties`.
● *CreateKeySpaceRequest*: Request to create a `keySpace`(akin to a table) in a specific `dataSpace` in a data management protocol, with a given set of `properties`.
● *ExecuteRequest*: Request to execute an operation on a specific `dataSpace` and `keySpace`.
The operation in question is specified through the abstract class `CommonOperation` which can be instantiated with a specific operation type (i.e., `BlockchainOperation`, `PayloadOperation`, etc.).
● *DeleteDataspaceRequest*: Request to delete a `dataSpace`.
● *DeleteKeySpace*: Request to delete a `keySpace` in a `dataSpace`.

### 3.6.2 Replies

● *CreateReply*: Generated in response to a `CreateDataspaceRequest` or `CreateDataspaceRequest`, with the status of the operation.
● *ExecuteReply*: Generated in response to a `ExecuteRequest`, with the status of the operation and the response data (e.g., a `payload`).
● *DeleteReply*: Generated in response to a `ExecuteRequest,DeleteKeySpace or DeleteDataSpace`, with the status of the operation.
● *NotSupportedReply*: Generated in response to a request for an operation that is not implemented in the underneath data management protocol.

### 3.6.3 Abstractions

- Arboreal *adapter*
- Cassandra *adapter*
- C3 *adapter*
- Engage *adapter*
- Blockchain-based *adapter*

### 3.6.4 Usage

To use the aforementioned classes, a user must first create (or use a previously created) Java project with the Babel-core and the Babel-ProtocolCommons dependencies.
With this, to make use of the provided abstractions, while developing its data management protocol (or creating an adapter as detailed in Section 2.c.iii) a user must incorporate them with the inter-protocol communication functionality offered by Babel, namely, by providing the request handlers in conformation with the API, and by sending information through *sendReply* and *triggerNotification* methods of Babel.

A few examples of these implementations can be found in babel-datareplication-adapters.

## 3.7 Epidemic Global View in Babel

Repository:
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/protocols/epidemicglobalview

A prototype of the epidemic global view membership protocol described in Section 2.2.2 is provided at the public repository identified above as a babel protocol, that implements the generic (and simplified) interface for membership protocols that we also described above on Section 3.4 (and that is, itself, a simplification of the one discussed in Section 2.2.1.

The prototype follows the pseudo-code presented before with some small adjustments, for instance, and since our implementation relies on another membership service that is based on partial-view of the system, the protocol only starts its periodic action to broadcast a ALIVE notification after the underlying membership service issues a notification that at least one other neighbour exists. This has the advantage that in a system with a single node active, the protocol does not consume any CPU or bandwidth.

We expect that most of the applications that will use this protocol will rely on the notifications of NeighborUp and NeighborDown instead of using the GetNeighborsSample request to collect random samples of elements in the system.

The protocol can be integrated into a babel application easily, by importing its maven dependency as detailed in the public repository.

## 3.8 Arboreal: Extending Data management from Cloud to Edge leveraging Dynamic Replication

Repository:
https://codelab.fct.unl.pt/di/research/tardis/wp6/public/arboreal

A fully working prototype implementation of Arboreal is available at https://codelab.fct.unl.pt/di/research/tardis/wp6/public/arboreal . The code is divided into 5 protocols, each with its own responsibilities, which interact with each other via message-passing. The protocols are:

- *hyparflood.HyParFlood.kt:* The HyParFlood protocol is a gossip-based protocol that uses HyParView to maintain a fully connected network overlay across all nodes. It uses a flooding mechanism that allows nodes to disseminate arbitrary information to all other nodes in the network. In our implementation, this information consists in the geographical location of the node.

- *manager.Manager.kt*: The Manager protocol controls the dynamic creation of the control tree. It uses a configurable heuristic, leveraging on the information collected by the HyParFlood protocol, to decide which node to use as parent, forming the initial control tree in a decentralised manner.
- *tree.Tree.kt*: The Tree protocol is responsible for the core features of Arboreal, it propagates and applies operations while enforcing causal consistency, handles the dynamic partial replication and is responsible for healing the tree after node failures.
- *storage.Storage.kt*: The storage protocol stores the data objects themselves (the storage engine can be changed; we use a simple in-memory key-value store). It also serves as the interface between the Tree protocol and the ClientProxy, resorting to the Tree to request data objects from other nodes when needed.
- *proxy.ClientProxy.kt*: The client proxy is simply the interface for the client to interact with the system. It maintains a list of clients, and forwards requests to the Storage protocol.

The Config.kt file contains all the configurable parameters of the application, including their default values. To launch a node, only a few of those are mandatory:

- "interface" or "address": the network interface or ip address used to listen for connection from other nodes (only one is required).
- "hostname": a domain name (name of the container when using docker networks) or ip address by which this node is reachable by other nodes.
- "region": an arbitrary string that represents a geographical region (each region forms its own control tree)
- "datacenter": the "hostname" of the node that serves as datacenter (i.e., root of the tree) for this node's region. Used as a contact point for joining the regional HyParFlood overlay. If a node's "datacenter" is the same as its "hostname", the node acts as the tree root.
- "location_x" and "location_y": coordinates of this node, affect the layout of the control tree.
- "tree_builder_nnodes": the number of nodes required to be part of the overlay before the control tree starts being formed.

To run the software a Java 8 installation is required as well as the installation of the Maven tool, required for performing the build and packaging operations. Therefore, before running the software a mvn compile package command should be executed, from the root of the project, to build the required jar file. Then, to launch a node, from the "deploy" folder, execute the following command:

```
java -DlogFilename=${log_file} -jar tree.jar param1=foo param2=bar
```

Where "log_file" is a path to the location of the application logs, and any number of parameters can be passed in the format <param>=<value>. Configuration parameters can be set both in the properties.conf file, or in the launch command, with the later overriding the former.

## 3.9 PotionDB: Strong Eventual Consistency under Partial Replication

Repository:
https://codelab.fct.unl.pt/di/research/tardis/wp6/public/potionDB

PotionDB's  current prototype can be found at the publicly accessible TaRDIS repository located in https://codelab.fct.unl.pt/di/research/tardis/wp6/public/potionDB. The code is split into 6 main components. Components interact with each other through message-passing by using Go's channels. All components of PotionDB scale with the number of CPU cores by relying on Go's goroutines and avoiding locking. PotionDB's components are as follows:

- main.Protoserver.go: This component provides the interface for clients to interact with PotionDB. The current implementation offers a Protobuf interface that extends AntidoteDB's Protobuf interface with extra features such as partial reading. PotionDB's working is agnostic to the Protobuf interface and thus could be replaced with another interface (e.g., REST, JSON, etc).
- antidote.TransactionManager.go: Implements PotionDB's transactional protocol, ensuring clients see a consistent view of the database and that updates evolve the state correctly. It is responsible for coordinating the partitions of Materializer and ensuring the correct generation of new, causally consistent, snapshots.
- antidote.Materializer.go: Implements the data storage portion of PotionDB. It is responsible for storing the objects, managing the available versions of each object and ensuring reads are applied on the correct version and updates generate new object versions consistently. The Materializer is partitioned in order to allow concurrent access to unrelated objects and scale with the cores of the CPU.
- antidote.Replicator.go: Handles the replication logic of PotionDB. It implements partial replication, ensuring that, for each transaction, each server only receives the operations of the transaction that are relevant for said server, while still ensuring each server can advance its state correctly.
- antidote.RemoteConnection.go: Handles the communication between PotionDB and RabbitMQ. We leverage RabbitMQ to propagate each transaction's updates to the relevant servers.
- CRDT: The crdt package implements the data types supported by PotionDB, in the form of CRDTs (Conflict-Free Replicated Data Types). Note that our protocols are not dependent on CRDTs and thus other data types could be used, as long as they ensure state convergence under weak consistency.

A Docker image is provided for ease of running PotionDB. To obtain a pre-built version of PotionDB, execute the following command from the root of the repository:

```
docker pull andrerj/potiondb
```

Alternatively, to build directly from the source:

```
docker build . -t mypotiondb
```

To run potionDB:

```
docker run -p 8087:8087 -p 5672:5672 --name potiondb andrerj/potiondb
```

If using a build directly from the source, replace "andrerj/potiondb" with "mypotiondb".
A new instance of PotionDB will be started on ports 5672 and 8087. Clients connect to port 8087, while other servers connect to 5672 for replication purposes. The server is ready to serve requests as soon as the following message appears:

```
PotionDB started at port 8087 with ReplicaID 23781
```

Note that ReplicaID will vary between executions as it is randomly generated. To stop PotionDB, execute the following command:

```
docker stop -t 1 potiondb
```

To run PotionDB without docker, both Go and a RabbitMQ installation are required. Further instructions can be found at https://codelab.fct.unl.pt/di/research/tardis/wp6/public/potionDB .

PotionDB can be parameterized by supplying a configuration file. The file configs/singlePC/docker/SingleServer/sampleConfig.cfg contains an example configuration file. The most relevant configurations are:

- protoPort: the network port to which clients connect to;
- buckets: a list of buckets (topics) that this server replicates, separated by a space. The wildcard '*' can be used to replicate all buckets.
- localRabbitMQAddress: ip:port where this server's RabbitMQ instance is running on. This should only need to be changed if running PotionDB outside of docker.
- localPotionDBAddress: ip:port where PotionDB is running and reachable. If running locally on Docker, the ip can be replaced by the name of the docker container.
- remoteRabbitMQAddresses: list of ip:port of the RabbitMQ instances of other PotionDB servers to connect to for replication purposes.
- nPartitions: number of partitions of the Materializer. This may affect PotionDB's performance. It should not be higher than the number of CPU cores available for PotionDB to use.

To use a new configuration file with PotionDB in Docker, build PotionDB from the source and then run PotionDB as follows:

```
docker run -p 8087:8087 -p 5672:5672 --name potiondb andrerj/potiondb
-e CONFIG=/go/bin/configs/path_to_your_config_folder
```

If running PotionDB without Docker, the config folder can be supplied with `–config=path_to_your_config_folder`.

## 3.10 Integration of Storage Solutions into the TaRDIS Ecosystem (Blockchain, C3, Engage)

Repository:
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel/babel-datareplication-adapters

To face the complexity of integrating generic storage solutions into external ecosystems, we designed a set of *adapters* that aim to provide client-side implementations so that applications can interact with their desired storage solution. To enable this, we leveraged Babel inter-communication protocol functionalities and used the previously discussed abstractions in Babel, to offer a common interface for interacting with the data management layer (more details in Section 2.7).

The adapters prototype can be found at babel-datareplication-adapters.
At the moment we offer adapters for Arboreal, C3, Cassandra, HyperLedger Fabric (Blockchain), and Engage.

### 3.10.1 Usage

Choose one of the implemented adapters by integrating its respective class (located at `pt.unl.fct.di.novasys.babel.adapters.`***solutionName***) into the protocol stack. This is done by using the Babel initializer and passing the corresponding properties needed to initialise the protocol (depending on the storage solution being used).

The communication with the adapter is made through Babel `sendRequest` and `receiveReply` abstractions. The formats for interacting with these abstractions are common for all adapters, and their semantics are further detailed in `babel-protocolcommons` under the `datamanagement` package.

It is important to notice that the corresponding storage solution must be deployed in an infrastructure of the choice of the developer to enable the interaction with the client-side adapter being used by the application (further details can be found in the repositories of the available solutions in TaRDIS toolbox, under external-tools).

## 3.11 DISTRIBUTED MANAGEMENT OF CONFIGURATION BASED ON NAMESPACES

Repository:
https://codelab.fct.unl.pt/di/research/tardis/wp6/public/configuration-management

The repository with the software is available on the link provided above. The prerequisites for running software are installed containerized tools Docker and Docker Compose. For Microsoft Windows users, the one more prerequisite must be met, a Unix-like environment and command-line interface (e.g. git bash for windows, Cygwin). All mentioned tools are open source.

Upon pulling the source code from the repository, and successful installation of the open source tools, users need to navigate in the *tools* folder. Inside this folder, we provided two scripts with the intention to simplify starting and stopping of all developed services, and all connected components.

Script *start.sh*, will build and run all necessary components, tie them into the network and it will allow users to test the entire tool. Navigate the terminal into the project repository, and using command *cd tools*, navigate inside the *tools* folder. By typing *start.sh* and pressing *Enter*, the entire system will start the build and running process. During this process, all required Docker images will be pulled on the testing machine, while Docker Compose will provide network between started services and all other connected components. Please, be aware that this process might take some time. For Microsoft Windows users, who are using Unix-like environments and command-line interfaces (e.g. git bash for windows, Cygwin), please be aware that the process to run the software is almost the same, except that after navigation to the *tools* folder, they start the script with *./start-windows.sh* command.

To stop the software from running, please open a new terminal or Unix-like environment and command-line interface (e.g. git bash for windows, Cygwin) for Microsoft Windows users, if the previous terminal window is not accessible due to the running software and log output from the system. Again, navigate to the software folder and then to folder tools, type *./stop.sh* and pressing Enter should stop the entire service and all its components. Microsoft Windows users need to run the ./stop.sh command. Again, please be aware that the script will stop the entire software and all its connected components, but it will take some time to do that.

In the repository, users can find the detailed documentation of what services, endpoints and functionalities are available at the moment, and how to use them. The repository also contains detailed explanation on how to format data, what data format the services expect, but also examples of data response from every service.

## 3.12 Telemetry Acquisition for Decentralised Systems

| Repository: https://codelab.fct.unl.pt/di/research/tardis/wp6/public/babel-core-metrics/ |
|---|

Babel, an integral part of several components comprising the TaRDIS toolbox, is a framework designed to simplify the process of developing distributed applications and protocols, achieving this purpose by handling low-level aspects of distributed systems programming, such as concurrency, message passing and timer management. However, it does not allow developers to easily collect telemetry about their developed applications.

To correct this shortcoming, we are developing a metrics collection system using the aforementioned architecture that is to be integrated into the Babel framework, this integration is done by generating a new release of the Babel framework that includes the metrics collection and exporting system.

The code for the solution is present in the TaRDIS public repository, which can be accessed here: https://codelab.fct.unl.pt/di/research/tardis/wp6/public/babel-core-metrics/.
The relevant code pertaining to the solution is present inside the metrics package.

To use the Babel framework to develop a distributed application or protocol you must add it as a Maven dependency to your project as instructed in the README presented in the repository pointed by the above link.

Examples on how to instrument a protocol to collect metrics and initialise and configure an exporter to export the collected metrics are also presented in the same README, along with how to configure the exporters.

# 4 STATE OF THE ART REVISION

## 4.1 DECENTRALISED MEMBERSHIP AND COMMUNICATION PRIMITIVES

There are many decentralised membership and communication primitives proposed in the literature, as discussed throughout this report. In the peer-to-peer literature there are two main classes of membership abstractions: peer sampling services [100] and overlay networks [25,22,26]. Peer sampling services are commonly considered as the substrate to support scalable gossip-based protocols, where nodes only need to interact with random samples of other nodes in the system. In many cases subsequent interactions benefit from targeting different nodes, as it is the case with anti-entropy protocols [38]. Overlay networks, as the name implies, define a logical network on top of other networks - many times an IP network - that nodes in a system can use to coordinate their interactions. In this report we unify these concepts by making the implicit observation that the implementation of both abstractions relies on each element of the system maintaining local information about a fraction of other elements in the system, the so-called partial views [22]. We go a step further and unify this concept with that of distributed hash tables (DHTs) [39,40,41,35] that are commonly referred to as a decentralised solution for storage or application-level routing. While we agree with the observation that these are suitable for these operations, we note that the routing tables maintained by each node are themselves partial views, which can be used to obtain samples of nodes in the system, for instance to govern anti-entropy protocols, or to define a logical network across nodes in a system. These observations have been previously done by Leitão in his PhD thesis [19].

Contrary to works in the literature, in this report we proposed a decentralised membership service that provided to each node in the system a global view of the system membership that is eventually correct. This breaks with the common approach to rely on partial membership information, which avoids a significant upkeep overhead (both in terms of communication and CPU). While we recognise that the overhead of this membership abstraction is higher than those based on partial views, we argue that such an abstraction can be used to collect information or provide user feedback about the evolution of the system in medium-scale swarm systems, such as the factory setting put forward by Actxy.

There is an engineering contribution enclosed in this report in relation to the extensive state of the art in decentralised membership and communication primitives and protocols, which is the definition of a common API for these classes of solutions, and implementations (in the Babel framework) that can be used interchangeably across swarm applications. While the literature in these domains is vast, typically each proposal is presented in an independent way from others, and existing prototypes are not provided (when they are provided) in a way that simplifies their usage across different applications. Even considering p2p frameworks such as the libp2p framework, decentralised protocols implementations there, such as Kademlia [35] are very hard to extract and be reused in some other decentralised application. We note however, that this engineering contribution is only feasible because we identify what should be the generic interface of these protocols (that aggregate a wide family of decentralised protocols from the literature), which is itself a contribution achieved by TaRDIS.

## 4.2 DISTRIBUTED DATA MANAGEMENT SYSTEMS

Distributed data management systems are one of the key services supporting large scale systems, as applications need to store their data reliably. A very large number of systems have been designed, which can be broadly classified as being either strongly or weakly consistent.

Funded by the European Union

Strong consistency systems [60, 95, 61, 62] provide a total order for transactions and are easier to work with. However, coordination between replicas is required, leading to high latency in geo-distributed and edge scenarios and compromised availability under network partitions. By relaxing consistency guarantees, weak consistency systems [63, 64, 66, 68] can provide low latency, high throughput and better fault tolerance. However, concurrency conflicts make those solutions harder to work with. Causal consistency alleviates this problem; however anomalies can still be observed even with cross-object causality [67]. Causal+ consistency extends causal consistency by enforcing eventual convergence. We now overview the state of the art that is closer to the works we are developing in the context of this project.

**Data Replication with Causal+ Consistency:** Multiple solutions for data replication with causal+ consistency have been proposed in the past. However, a great majority of them assume data centre deployments, with a limited number of replicas, and without supporting partial replication. Such solutions include COPS [64] and Eiger [82], which track causality using explicit dependencies. Other solutions employ vector clocks, resulting in their metadata growing proportionally to the number of replicas. These include, Orbe [85] and Cure [68], among others. ChainReaction [66] uses vector clocks with a size associated to the number of data centres and not individual replicas, but still, this results in excessive metadata overhead in large scale edge scenarios. As we showed in our evaluation, the metadata overhead of these solutions is prohibitive in edge scenarios. Some Causal+ replication solutions have been proposed that aim to limit metadata overhead, by using fixed-sized metadata. Examples include Saturn [76] and GentleRain [86]. However, regardless of the metadata size, these solutions do not consider edge deployments, and lack crucial features such fault-tolerance handling mechanisms and partial replication, making them unusable in edge scenarios.

**Data Replication in the Edge:** With the rise of edge computing, multiple solutions for data management on the edge have been proposed. Solutions like PathStore [88], DataFog [87], and CloudPath [89] do not provide any consistency guarantees. On the other hand, some solutions attempt to provide strong consistency. However, due to the highly geo-distributed nature of the edge, providing global strong consistency is impractical. As such, these solutions limit those guarantees to individual smaller, well-connected groups of nodes, relaxing consistency across groups. Examples include EdgeKV [90], FogStore [91], Colony [77], and DAST [92]. Despite being mitigated by the smaller size of groups and high connectivity assumptions, these solutions suffer from lack of availability and fault-tolerance, as most strong consistency solutions.

As for causal+ consistency on the edge, few solutions have been proposed. However, all of them have limitations that reduce their applicability in edge environments. Gesto [93] relies on a single centralised component in each region to enforce causal consistency, while Colony [77] (which also supports global transactional causal consistency along with strong consistency within groups) relies on the data centre to validate all transactions. In these solutions, edge nodes (or groups in the case of Colony) cannot cooperate, requiring the cloud to mediate all interactions and disallowing nodes from progressing independently of the cloud. As we showed in our evaluation, this negatively impacts not only performance but also edge-specific aspects such as dynamic replication and client mobility. Engage [94] provides decentralised global causality in the edge but relies on vector clocks which greatly limits its scalability.

## 4.3 DECENTRALISED MONITORING

Monitoring tools play an essential role in getting a grasp on system behaviour. Their task is becoming ever so challenging in ephemeral and heterogeneous environments involving both the cloud and edge. Dynamic placement and resource allocation of distributed workloads, as well as different packaging and isolation strategies, all contribute to poor observability. Combined with resource-constrained and heterogenous edge infrastructure, they call for a comprehensive monitoring solution. To provide valuable insight and enable timely reactions, these should bundle a variety of capabilities, including data collection, storage, aggregation, querying, visualisation and alerting mechanisms.

A couple of vendor-agnostic standards have emerged to facilitate integration of diverse data sources and types. *OpenMetrics* [8] provides a format for representing metrics data, while *OpenTelemetry* [9], aside from defining standard formats for logs, metrics and traces, offers a rich ecosystem of APIs and SDKs to instrument applications. These should stand as building blocks of observability platforms for the future. No such open-source platform has gained as much traction as *Prometheus* [10] has. It is a time series database with a rich query language, designed primarily for metrics. Its features extend beyond that as it has a tight integration with *Grafana* for visualisation and *Alertmanager* [11] for alerting. As such, it's found its place in numerous deployments and is natively supported by *Kubernetes* for cluster monitoring. However, as it doesn't focus on logs, traces and other observability data, it doesn't provide a full overview of the system's state. It can be incorporated in a tool stack that in the end produces sufficient insight, but this puts a significant operational burden on users. Also, general-purpose stacks like those have no built-in primitives for working in a cloud-edge setting.

Our objective is to develop an observability platform that can collect, aggregate, transform and store diverse monitoring-related data from heterogeneous sources and with respect to resource availability. Users will be provided with a unified interface of system and application state progression over time, which will allow for informed decision making. If the decision process is to be automated, our platform offers an API for machine learning models to be both trained on and put in the role of decision makers. As volume of telemetry generated at the edge can easily exceed available storage and computing capacity there, we'll define strategies for its migration to more resource-abundant environments, where heavier processing can occur. However, naively transferring all data would introduce high latency and potentially raise data privacy concerns. For those reasons, data is to be filtered or transformed before relocation. This way, the weighted cost of processing data locally versus remotely can be taken into account. Data that is not to leave its origin can still be analysed by authorised individuals and utilised for federated learning purposes.

## 4.4 Decentralised System Management

System management operations can be divided into application and underlying infrastructure management. Both can be executed manually, but such approach is not fit for any slightly more complex system, let alone those relying on heterogeneous and dynamic cloud and edge models. With no configuration workflow established, human-induced configuration drifts are a likely consequence. System state easily starts deviating from the desired state if there is no straightforward and automated way of enforcing and tracking configuration changes.

For those reasons, substantial efforts have gone into automating infrastructure management. Many tools have been introduced under the umbrella term Infrastructure as Software (IaS), all with the aim of ensuring consistent and repeatable configuration. Using them, the risk of misconfiguration gets minimised, as many well-established software practices can be employed, including reusability, versioning, and testing. The method of specifying intent to these tools can be imperative or declarative. *Terraform* [12], *Polumni* [13] and *CFEngine* [14], being representatives of the declarative approach, expect the user only to state the desired outcome, while they internally deal with the intricacies of the target infrastructure. This additional layer of abstraction turned out immensely useful in complex multi-cloud and hybrid deployments. However, when more control over the underlying mechanism is needed, imperative-oriented tools like *Chef* [15], *Ansible* [16] and *Puppet* [17], are a favourable option. With them, users explicitly code the steps to be taken, in some domain-specific or a general-purpose language. While both declarative and imperative tools complimentary contribute to the automated infrastructure management, they all lack native support for cloud-extending paradigms utilising geo-distributed resources at the edge.

Building on top of [3, 4], our work aims to support distributed cloud configuration on both the infrastructure and application level by adapting existing approaches to this and other novel cloud-edge environments. As the IaS concept has already yielded numerous benefits, we plan to develop a comprehensive Configuration-as-Software solution in a similar fashion. A declarative API will hide the complexities irrelevant to the end user, while exposing primitives for optimally exploiting the resource variability and geographical distribution of the infrastructure. To enhance robustness and efficiency of the entire system, we will define strategies for decentralised management and propagation of configuration.

As human-centred applications of the future are driving the need for deployment models more complex than ever, it is crucial to provide developers and platform engineers with reliable tools and familiar APIs for interacting with such environments. Only then can these applications reach their full potential. Our solution presents one of many steps leading to that goal.

# 5 Publications and Dissemination Activities

## 5.1 Publications

Some of the results shown here have been published in scientific conferences. Publications produced during the reported period related with the activities reported here include:

- Data Management for Mobile Applications Dependent on Geo-Located Data. N. M. Santos, L. M. Silva, J. Leitão, and N. Preguiça. Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC'23) colocated with EuroSys, May 8, Rome, Italy, 2023.

- Studying the Workload of a Fully Decentralized Web3 System: IPFS. Pedro Á. Costa, João Leitão, and Yiannis Psaras. Proceedings of the 23rd International Conference on Distributed Applications and Interoperable Systems (DAIS'23) part of the DisCoTec (International Federated Conference on Distributed Computing Techniques), June 19-23, Lisboa, Portugal, 2023.

- Kovacevic, I., Stojkov, M., Simic, M. (2024). Authentication and Identity Management Based on Zero Trust Security Model in Micro-cloud Environment. In: Trajanovic, M., Filipovic, N., Zdravkovic, M. (eds) Disruptive Information Technologies for a Smart Society. ICIST 2023. Lecture Notes in Networks and Systems, vol 872. Springer, Cham. https://doi.org/10.1007/978-3-031-50755-7_45

- Maksimović, V., Simić, M., Stojkov, M., Zarić, M. (2024). Task Queue Implementation for Edge Computing Platform. In: Trajanovic, M., Filipovic, N., Zdravkovic, M. (eds) Disruptive Information Technologies for a Smart Society. ICIST 2023. Lecture Notes in Networks and Systems, vol 872. Springer, Cham. https://doi.org/10.1007/978-3-031-50755-7_44

## 5.2 Dissemination Activities

In the reported period, WP6 has contributed to a few dissemination activities of TaRDIS. In particular, WP6 has led the efforts to promote the Babel framework, that we decided to use as a tool to build reference implementations of the technology developed in WP6 at the International Federated Conference on Distributed Computing Techniques (DisCoTec), that took place in Lisbon in June 2023.

The tutorial, entitled "Implementing and Evaluating Distributed Protocols with Babel", showcased how Babel could be used to build distributed applications, and decentralised applications, and referred to the TaRDIS project.

The materials for this tutorial, done by João Leitão, Pedro Ákos Costa, and Pedro Fouto (all from NOVA) can be found here: https://github.com/pfouto/babel-tutorial

# 6 RELATIONSHIP WITH OTHER TECHNICAL WORK PACKAGES

We now detail some of the interactions and relationships between the activities of WP6 and the other technical work packages (WP3, WP4, WP5, and WP7).

## 6.1 WP3

WP6 has been responsible for proposing the initial generic APIs for abstractions such as Membership management, communication primitives, distributed data management, and monitoring and configuration management.

This initial specification that was reported in Deliverable 3.1 [58], has been revised through the process of devising adaptors for existing storage systems, and through the implementation of distributed protocols reported here. This continuous revision of APIs will continue throughout the project as we consider different implementations of these abstractions, but also as detailed in the previous sections, as we consider complementary aspects such as security.

Additionally, some of the tools and artefacts being produced by WP6 will be integrated into the IDE being developed in the context of WP3. Although not reported in this deliverable, the TaRDIS consortium already has produced a pilot integration of the Babel framework into the IDE being developed as part of the activities of WP3. The fundamental ideal of this integration is to simplify and guide the developer when using Babel to develop a swarm protocol or application, by providing templates, and in the future, wizards to assist in the integration of artefacts (such as concrete distributed protocols).

## 6.2 WP4

While WP6 is building components that can be used - in the context of the TaRDIS toolbox - to build swarm applications, WP4 will provide tools that validate the correctness of these applications. To this end, WP6 will have to provide (formal) properties for the components developed by WP6, such that these properties can be taken into consideration when trying to show the correctness (or not) of a particular application that uses them.

While in the work conducted by WP6 we have not yet produced these formal specifications, this will have to be addressed in the next cycle of development of the TaRDIS project.

## 6.3 WP5

Collection of telemetric information is one of the crucial things in complex systems, because it gives the users insight into what is going on with the entire system and/or its applications. Designing a platform to collect, aggregate, transform and store diverse monitoring-related data from heterogeneous sources and with respect to resource availability, will allow for informed decision making. The collected data could be further used for automated decision making processes. To allow this, the platform will offer an API for machine learning (ML) models to be both trained on and put in the role of decision makers.

The collected data will be stored in the platform in the time-series manner (i.e. data points with time). Through the specifically designed API, the platform will offer its collected data to the agents who will be trained on these data points. The only thing that API requires from the agent is a point in time, when the agent contacted the platform last time. With this, machine learning models can be trained at their own pace, and agents cannot miss or skip newly added data points, even if the agent crashes, or goes offline for some period of time.

To this end, a significant coordination and collaboration between WP5 and WP6 is needed, since: (i) Task 6.1 that aims to provide the decentralised membership and communication primitives for the Tardis platform will be directly exploited by the Federated Learning (FL) framework and algorithms developed in WP5. In specific, both the centralised (including a server/aggregator and several clients) and decentralised (including only clients as swarm members) FL frameworks of Task 5.1 necessitate reliable peer-to-peer communication, enabling high availability of the swarm agents and low-latency and energy-efficient communications (Task 5.3); (ii) Similarly, Task 6.2 covers the design aspects of decentralised data management and replication schemes that the ML algorithms will leverage, since historical datasets that are stored locally are often required to enable ML model re-training capabilities and improved collaborative intelligence of the swarm; (iii) Finally, Task 6.3 targets on the one hand to develop the decentralised monitoring scheme and on the other hand to design the reconfiguration management framework. Concerning the former aim, the monitoring of decentralised telemetry data will include the performance of local ML models and the data analysis can be also assisted by ML algorithms. Regarding the latter objective, the reconfiguration of application components and computational needs of resources in runtime can be orchestrated with the assistance of AI/ML methods, using for instance the suggestions of Reinforcement Learning agents (Task 5.2).

## 6.4 WP7

WP7 is responsible for coordinating the use case implementations and conducting their experimental validation. Related with this activity, WP6 has produced prototypes (and reference implementations) for components that can be used in the development of use cases. WP6 has also conducted initial experimental validations of some of these components.

Finally, and while not explicitly reported here, WP6 has also conducted some work in gathering relevant information to conduct experimental assessment of swarm systems, for instance by measuring the workload characteristics of popular decentralised systems such as IPFS [103], which can inform some of the experimental validation work to be conducted in the future as part of the efforts of WP7.

# 7 CONCLUSIONS

This report provides information about the main activities conducted in the three tasks of the WP6 of the TaRDIS project in the first 14 months of the project. The report also includes pointers and brief descriptions for the main artefacts produced by WP6 in the reported period, that serve as prototypes of some of the solutions being developed by the project and reference implementations. This period of the project has been dedicated significantly to analysing and identifying the requirements of the industrial use cases, which allowed WP6 to conduct some exploratory work and development for the most fundamental building blocks towards supporting swarm applications. In the future, we will start to address challenges such as security, and coalescing the results of the project into concrete tools that will integrate the TaRDIS toolbox.

As an additional note, WP6 has contributed to define requirements to the TaRDIS toolbox that have been reported on Deliverable 2.2 [96] that were (mostly) derived from requirements of the use cases. Out of these, the artefacts or results produced by WP6 and mentioned here address requirements: RF-WP6-MA-05, RF-WP6-MA-12, RNF-WP6-MA-14, RNF-WP6-MA-15, RF-WP6-CP-19, RF-WP6-CP-20, RNF-WP6-CP-23, RF-WP6-SA-29, RNF-WP6-SA-32, and RF-WP6-TA-34.

Funded by the European Union

## REFERENCES

[1] Burns, B. Grant, D. Oppenheimer, E. A. Brewer, and J. Wilkes, “Borg, omega, and kubernetes,” Commun. ACM, vol. 59, no. 5, pp. 50–57, 2016. [Online].Available: https://doi.org/10.1145/2890784

[2] Kelsey Hightower, Brendan Burns, and Joe Beda. 2017. Kubernetes: Up and Running Dive into the Future of Infrastructure (1st. ed.). O'Reilly Media, Inc.

[3] Simić, Miloš & Sladic, Goran & Zarić, Miroslav & Markoski, Branko. (2021). Infrastructure as Software in Micro Clouds at the Edge. Sensors. 21. 7001. 10.3390/s21217001.

[4] Simić, Miloš & Prokić, Ivan & Dedeić, Jovana & Sladic, Goran & Milosavljević, Branko. (2021). Towards Edge Computing as a Service: Dynamic Formation of the Micro Data-Centers. IEEE Access. PP. 1-1. 10.1109/ACCESS.2021.3104475.

[5] Fitzgerald, N. Forsgren, K.-J. Stol, J. Humble, andB. Doody, “Infrastructure is software too!” SSRN Elec-tronic Journal, 01 2015.

[6] G. Pallavi, D. P. Jayarekha, Multi-tenancy in saas–a comprehensive survey, International journal of scientific and engineering research 7 (7) (2014) 680.

[7] A. Makroo, D. Dahiya, A systematic approach to deal with noisy neigh- bour in cloud infrastructure, Indian Journal of Science and Technology 9 (19) (May 2016). doi:https://doi.org/10.17485/ijst/2016/v9i19/89211.

[8] OpenMetrics, Cloud Native Computing Foundation, https://openmetrics.io/, Accessed January 2024.

[9] OpenTelemetry, Cloud Native Computing Foundation, https://opentelemetry.io/, Accessed January 2024.

[10] Prometheus, Cloud Native Computing Foundation, https://prometheus.io/, Accessed January 2024.

[11] AlertManager, Cloud Native Computing Foundation, https://prometheus.io/docs/alerting/latest/alertmanager/, Accessed January 2024.

[12] Terraform, HashiCorp, https://www.terraform.io/, Accessed January 2024.

[13] Polumni, Polumni, https://www.pulumi.com/, Accessed January 2024

[14] CFEngine, CFEngine, https://cfengine.com/, Accessed January 2024.

[15] Chef, Progress Software, https://www.chef.io/, Accessed January 2024.

[16] Ansible, Red Hat Software, https://www.ansible.com/, Accessed January 2024.

[17] Puppet, Puppet, https://www.puppet.com/, Accessed January 2024.

[18] A. S. T. Maarten van Steen. Distributed Systems. 4th ed. distributed-systems.net, 2023. (Chapter 1).

[19] J. Leitão. “Topology Management for Unstructured Overlay Networks”. PhD thesis. Technical University of Lisbon, 2012.

[20] I. Clarke et al. “Freenet: A Distributed Anonymous Information Storage and Retrieval System”. In: Lecture Notes in Computer Science 2009 (2001-03). doi: 10.100 7/3-540-44702-4_4

[21] M. Conoscenti, A. Vetrò, and J. C. De Martin. “Peer to Peer for Privacy and Decentralization in the Internet of Things”. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). 2017, pp. 288–290. doi: 10.1109/ICSE-C.2017.60 (cit. on pp. 1, 2, 9).

[22] J. Leitão, J. Pereira and L. Rodrigues. “HyParView: a membership protocol for reliable gossip-based broadcast.” Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Edinburgh, UK, June, 2007.

[23] J. Leitão. Gossip-based broadcast protocols Master's Thesis, University of Lisbon, 2007.

[24] J. Leitão, J. Pereira, and L. Rodrigues. Gossip-Based Broadcast. In the Handbook of Peer-to-Peer Networking, X. Shen, H. Yu, J. Buford, M. Akon (Eds.), Springer 2010. ISBN: 978-0-387-09750-3

[25] Voulgaris, S., Gavidia, D., & Steen, M. (2005, June). CYCLON: Inexpensive membership management for unstructured P2P overlays. Journal of Network and Systems Management, 13(2), 197 – 217.

[26] Ganesh, A., Kermarrec, A.-M., & Massoulie, L. (2003, February). Peer-to-peer membership management for gossip-based protocols. IEEE Transactions on Computers, 52(2), 139 – 149.

[27] Rostami, H., & Habibi, J. (2007, May). Topology awareness of overlay P2P networks. Concurrency and Computation: Practice & Experience - Autonomous Grid, 19, 999 – 1021.

[28] Jelasity, M., Montresor, A., & Babaoglu, O. (2009, August). T-Man: Gossip-based fast overlay topology construction. Journal Computer Networks: The International Journal of Computer and Telecommunications Networking, 53(13), 2321 – 2339.

[29] J. Pouwelse et al. "The Bittorrent P2P File-Sharing System: Measurements and Analysis". In: vol. 3640. 2005-02, pp. 205–216. isbn: 978-3-540-29068-1. doi: 10.1007/11558989_19

[30] M. Conoscenti, A. Vetrò, and J. C. De Martin. "Peer to Peer for Privacy and Decentralization in the Internet of Things". In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). 2017, pp. 288–290. doi: 10.1109/ICSE-C.2017.60

[31] P. Poonpakdee, J. Koiwanit, and C. Yuangyai. "decentralised Network Building Change in Large Manufacturing Companies towards Industry 4.0". In: Procedia Computer Science 110 (2017). 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops, pp. 46–53. issn: 1877-0509. doi: https://doi.org/10.1016/j.procs.2017.06.113. url: https://www.sciencedirect.com/science/article/pii/S1877050917312929

[32] D. Lucke, C. Constantinescu, and E. Westkämper. "Smart Factory - A Step towards the Next Generation of Manufacturing". In: 2008-01, pp. 115–118. isbn: 978-1- 84800-266-1. doi: 10.1007/978-1-84800-267-8_23

[33] J. Qin, Y. Liu, and R. Grosvenor. "A Categorical Framework of Manufacturing for Industry 4.0 and Beyond". In: Procedia CIRP 52 (2016). The Sixth International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV2016), pp. 173–178. issn: 2212-8271. doi: https://doi.org/10.1016/j.procir.2016.0 8.005. url: https://www.sciencedirect.com/science/article/pii/S2212827 11630854X

[34] S. K. Routray et al. "Satellite Based IoT for Mission Critical Applications". In: 2019 International Conference on Data Science and Communication (IconDSC). 2019, pp. 1–6. doi: 10.1109/IconDSC.2019.8817030

[35] P. Maymounkov and D. Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: Peer-to-Peer Systems. Ed. by P. Druschel, F. Kaashoek, and A. Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65. isbn: 978-3-540-45748-0

[36] Frey, D., Guerraoui, R., Kermarrec, A.-M., Koldehofe, B., Mogensen, M., Monod, M., et al. (2009, November). Heterogeneous gossip. In Proceedings of the 10th ACM/IFIP/USENIX international conference on middleware (Middleware'09) (pp. 3:1 – 3:20). Urbana Champaign, Illinois, USA: Springer-Verlag New York, Inc.

[37] J. Leitão, J. Pereira, and L. Rodrigues. Epidemic Broadcast Trees. Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems, Beijing, China, October, 2007.

[38] J. Leitão, R. van Renesse and L. Rodrigues. Balancing Gossip Exchanges in Networks with Firewalls. Proceedings of the 9th International Workshop on Peer-to-Peer Systems (IPTPS '10), San Jose, CA, USA, 27 April, 2010.

[39] Stoica, I., Morris, R., Karger, D., Kaashoek, M., & Balakrishnan, H. (2001, September). Chord: A scalable peer-to-peer lookup service for Internet applications. In Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM'01) (pp. 149 – 160). San Diego, California, USA.

[40] Zhao, B., Huang, L., Stribling, J., Rhea, S., Joseph, A., & Kubiatowicz, J. (2004, January). Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications, 22(1), 41 – 53.

[41] Rowstron, A., & Druschel, P. (2001, November). Pastry: Scalable, decentralised object location, and routing for large-scale peer-to-peer systems. In Proceedings of the IFIP/ACM international conference on distributed systems platforms heidelberg (Middleware'01) (pp. 329 – 350). Heidelberg, Germany

[42] Jelasity, M., Montresor, A., & Babaoglu, O. (2009, August). T-Man: Gossip-based fast overlay topology construction. Journal Computer Networks: The International Journal of Computer and Telecommunications Networking, 53(13), 2321 – 2339.

[43] J. Leitão, J. P. Marques, J. Pereira, and L. Rodrigues. X-BOT: A Protocol for Resilient Optimization of Unstructured Overlays. Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems, Niagara Falls, New York, U.S.A., Sep, 2009.

[44] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. 1999. Bimodal multicast. ACM Trans. Comput. Syst. 17, 2 (May 1999), 41–88. https://doi.org/10.1145/312203.312207

[45] J. Pereira, N. Carvalho, R. Oliveira and L. Rodrigues, "Emergent Structure in Unstructured Epidemic Multicast," in 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07), Edinburgh, 2007 pp. 481-490. doi: 10.1109/DSN.2007.40

[46] J. Leitão, N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. On Adding Structure to Unstructured Overlay Networks. In the Handbook of Peer-to-Peer Networking, X. Shen, H. Yu, J. Buford, M. Akon (Eds.), Springer 2010. ISBN: 978-0-387-09750-3

[47] Liang, J., Ko, S., Gupta, I., & Nahrstedt, K. (2005, December). MON: on-demand overlays for distributed system management. In Proceedings of the 2nd conference on real, large distributed systems - volume 2 (WORLDS'05) (pp. 13 – 18). San Francisco, CA.

[48] M. Ferreira, J. Leitão and L. Rodrigues, "Thicket: A Protocol for Building and Maintaining Multiple Trees in a P2P Overlay," 2010 29th IEEE Symposium on Reliable Distributed Systems, New Delhi, India, 2010, pp. 293-302, doi: 10.1109/SRDS.2010.19.

[49] Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., Stoica, I. (2003). Towards a Common API for Structured Peer-to-Peer Overlays. In: Kaashoek, M.F., Stoica, I. (eds) Peer-to-Peer Systems II. IPTPS 2003. Lecture Notes in Computer Science, vol 2735. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-45172-3_3

[50] J.. Leitão and L. Rodrigues. "Overnesia: A Resilient Overlay Network for Virtual Super-Peers". In: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems. 2014, pp. 281–290. doi: 10.1109/SRDS.2014.40

[51] I. Gupta et al. "Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead". In: Peer-to-Peer Systems II. Ed. by M. F. Kaashoek and I. Stoica. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 160–169. isbn: 978-3-540-45172-3.

[52] Y. Chawathe et al. "Making Gnutella-like P2P Systems Scalable". In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. SIGCOMM '03. Karlsruhe, Germany: Association for Computing Machinery, 2003, pp. 407–418. isbn: 1581137354. doi: 10.1145/863955.864000. url: https://doi.org/10.1145/863955.864000

[53] I. Baumgart, B. Heep, C. Hübsch and A. Brocco, "OverArch: A common architecture for structured and unstructured overlay networks," 2012 Proceedings IEEE INFOCOM Workshops, Orlando, FL, USA, 2012, pp. 19-24, doi: 10.1109/INFCOMW.2012.6193490.

[54] R. Baldoni, M. Platania, L. Querzoni and S. Scipioni, "Practical Uniform Peer Sampling under Churn," 2010 Ninth International Symposium on Parallel and Distributed Computing, Istanbul, Turkey, 2010, pp. 93-100, doi: 10.1109/ISPDC.2010.25.

[55] M. Ripeanu. "Peer-to-peer architecture case study: Gnutella network". In: Proceedings First International Conference on Peer-to-Peer Computing. 2001, pp. 99–100. doi: 10.1109/P2P.2001.990433

[56] C. Tang, R. Chang, and C. Ward. “GoCast: gossip-enhanced overlay multicast for fast and dependable group communication”. In: 2005 International Conference on Dependable Systems and Networks (DSN’05). 2005, pp. 140–149. doi: 10.1109/DSN.2 005.52

[57] R. Melamed and I. Keidar. “Araneola: a scalable reliable multicast system for dynamic environments”. In: Third IEEE International Symposium on Network Computing and Applications, 2004. (NCA 2004). Proceedings. 2004, pp. 5–14. doi: 10.1109/NCA.2 004.1347755

[58] TaRDIS Consortium. “Deliverable 3.1: First Report on Programming Model and API“ February 2024.

[59] P. Fouto, P. Á. Costa, N. Preguiça, and J. Leitão. “Babel: A Framework for Developing Performant and Dependable Distributed Protocols”. In: 2022 41st International Symposium on Reliable Distributed Systems (SRDS). Los Alamitos, CA, USA: IEEE Computer Society, 2022-09, pp. 146–155. doi: 10.1109/SRDS55811.2022.00022.

[60] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google’s globally distributed database. ACM Transactions on Computer Systems (TOCS) 31, 3 (2013), 1–22.

[61] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data Center Consistency. In Proceedings 8th ACM European Conference on Computer Systems (EuroSys ’13). ACM, Prague, Czech Republic, 113–126.

[62] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. CockroachDB: The Resilient Geo- Distributed SQL Database. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 1493–1509.

[63] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan 1376 Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon’s Highly Available Key-value Store. In Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP ’07). ACM, Stevenson, Washington, USA, 205–220.

[64] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don’t Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP ’11). ACM, Cascais, Portugal, 401–416.

[65] Werner Vogels. 2009. Eventually Consistent. Commun. ACM 52, 1 (Jan. 2009), 40–44. https://doi.org/10.1145/1435417.1435432

[66] Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: a causal+ consistent datastore based on chain replication. In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13). Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/2465351.2465361

[67] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[68] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhong- miao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong semantics meets high availability and low latency. In 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). IEEE, 405–414.

[69] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. INRIA TR. A comprehensive study of convergent and commutative replicated data types. (2011).

[70] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. 1995. A Critique of ANSI SQL Isolation Levels. SIGMOD Rec. 24, 2 (may 1995), 1–10. https://doi.org/10.1145/ 568271.223785

[71] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In Proceedings of the Twenty-Third ACM Symposium on Operating

Systems Principles. 1502 385–400.

[72] Leslie Lamport. 1998. The Part-Time Parliament. ACM Trans. Comput. Syst. 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227. 279229

[73] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Hari- zopoulos, Nabil Hachem, and Pat Helland. 2018. The end of an architectural era: It's time for a complete rewrite. In Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker. 463–489.

[74] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang and W. Shi, "Edge Computing for Autonomous Driving: Opportunities and Challenges," in Proceedings of the IEEE, vol. 107, no. 8, pp. 1697-1716, Aug. 2019, doi: 10.1109/JPROC.2019.2915983.

[75] R. Bhardwaj et al., "Ekya: Continuous learning of video analytics models on edge compute servers," in 19th USENIX Symposium on Networked Systems Design and Implementation, 2022.

[76] M. Bravo, L. Rodrigues,and P. Van Roy,"Saturn: A distributed metadata service for causal consistency," in Proceedings of the Twelfth European Conference on Computer Systems, 2017.

[77] I. Toumlilt, P. Sutra, and M. Shapiro, "Highly-available and consistent group collaboration at the edge with colony," in Proceedings of the 22nd International Middleware Conference, 2021.

[78] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in ACM SIGOPS 28th Symposium on Oper. Syst. Principles, 2021.

[79] Kulkarni, S.S., Demirbas, M., Madappa, D., Avva, B., Leone, M. (2014). Logical Physical Clocks. In: Aguilera, M.K., Querzoni, L., Shapiro, M. (eds) Principles of Distributed Systems. OPODIS 2014. Lecture Notes in Computer Science, vol 8878. Springer, Cham. https://doi.org/10.1007/978-3-319-14472-6_2

[80] M. Belém, P. Fouto, T. Lykhenko, J. Leitão, N. Preguiça and L. Rodrigues, "Engage: Session Guarantees for the Edge," 2022 International Conference on Computer Communications and Networks (ICCCN), Honolulu, HI, USA, 2022, pp. 1-10, doi: 10.1109/ICCCN54977.2022.9868846.

[81] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[82] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (nsdi'13). USENIX Association, USA, 313–328.

[83] Pointcheval, D., & Sanders, O. (2015). Short Randomizable Signatures. Cryptology ePrint Archive, Paper 2015/525. https://eprint.iacr.org/2015/525

[84] L. Lamport. "Paxos Made Simple". In: ACM SIGACT News (Distributed ComputingColumn) 32, 4 (Whole Number 121, December 2001) (2001-12), pp. 51–58. url: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[85] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: scalable causal consistency using dependency matrices and physical clocks. In Proceedings of the 4th annual Symposium on Cloud Computing (SOCC '13). Association for Computing Machinery, New York, NY, USA, Article 11, 1–14. https://doi.org/10.1145/2523616.2523628

[86] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In Proceedings of the ACM Symposium on Cloud Computing (SOCC '14). Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2670983

[87] H. Gupta, Z. Xu, and U. Ramachandran, "DataFog: Towards a Holistic Data Management Platform for the IoT Age at the Network Edge," in Proceedings of the USENIX Workshop on Hot Topics in Edge Computing (HotEdge), Boston (MA), USA, Jul. 2018. https://www.usenix.org/conference/hotedge18/presentation/gupta

[88] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan

Puzhavakath Narayanan. 2018. Pathstore, A Data Storage Layer for the Edge. Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '18). Association for Computing Machinery, New York, NY, USA, 519. https://doi.org/10.1145/3210240.3210813

[89] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. 2017. Cloudpath: a multi-tier cloud computing framework. In Proceedings of the Second ACM/IEEE Symposium on Edge Computing (SEC '17). Association for Computing Machinery, New York, NY, USA, Article 20, 1–13. https://doi.org/10.1145/3132211.3134464

[90] Karim Sonbol, Öznur Özkasap, Ibrahim Al-Oqily, and Moayad Aloqaily. 2020. EdgeKV: Decentralized, scalable, and consistent storage for the edge. J. Parallel and Distrib. Comput., Vol. 144 (2020), 28--40.

[91] Harshit Gupta and Umakishore Ramachandran. 2018. FogStore: A Geo-Distributed Key-Value Store Guaranteeing Low Latency for Strongly Consistent Access. In Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems (DEBS '18). Association for Computing Machinery, New York, NY, USA, 148–159. https://doi.org/10.1145/3210284.3210297

[92] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21). Association for Computing Machinery, New York, NY, USA, 210–227. https://doi.org/10.1145/3447786.3456238

[93] N. Afonso, M. Bravo and L. Rodrigues, "Combining High Throughput and Low Migration Latency for Consistent Data Storage on the Edge," 2020 29th International Conference on Computer Communications and Networks (ICCCN), Honolulu, HI, USA, 2020, pp. 1-11, doi: 10.1109/ICCCN49398.2020.9209720.

[94] M. Belém, P. Fouto, T. Lykhenko, J. Leitão, N. Preguiça and L. Rodrigues, "Engage: Session Guarantees for the Edge," 2022 International Conference on Computer Communications and Networks (ICCCN), Honolulu, HI, USA, 2022, pp. 1-10, doi: 10.1109/ICCCN54977.2022.9868846.

[95] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: serializable, low-latency, geo-replicated transactions. Proc. VLDB Endow. 12, 11 (July 2019), 1747–1761. https://doi.org/10.14778/3342263.3342647

[96] TaRDIS Consortium. "Deliverable 2.2: Report on overall requirements analysis" February 2024.

[97] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The ϕ accrual failure detector," in Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004., 2004, pp. 66–78.

[98] "Netty framework," https://netty.io/.

[99] P. Fouto, J. Leitão, and N. Preguiça. Practical and Fast Causal Consistent Partial Geo-Replication. Proceedings of the 17th IEEE International Symposium on Network Computing and Applications (NCA 2018). November 1-3, 2018. Cambridge, USA.

[100] Jelasity, M., Guerraoui, R., Kermarrec, AM., van Steen, M. (2004). The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations. In: Jacobsen, HA. (eds) Middleware 2004. Middleware 2004. Lecture Notes in Computer Science, vol 3231. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30229-2_5

[101] "Actyx middleware", https://developer.actyx.com/ and https://github.com/Actyx/Actyx

[102] "Actyx machine-runner", https://www.npmjs.com/package/@actyx/machine-runner and https://github.com/Actyx/machines

[103] Pedro Á. Costa, João Leitão, and Yiannis Psaras. Studying the Workload of a Fully Decentralized Web3 System: IPFS. Proceedings of the 23rd International Conference on Distributed Applications and Interoperable Systems (DAIS'23) part of the DisCoTec (International Federated Conference on Distributed Computing Techniques), June 19-23, Lisboa, Portugal, 2023.

[104] K. Jenkins, K. Hopkinson and K. Birman, "A gossip protocol for subgroup multicast," Proceedings 21st International Conference on Distributed Computing Systems Workshops, Mesa, AZ, USA, 2001, pp. 25-30, doi: 10.1109/CDCS.2001.918682.
[105] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. 2001. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In Proceedings of the Third International COST264 Workshop on Networked Group Communication (NGC '01). Springer-Verlag, Berlin, Heidelberg, 30–43.
[106] “Banyan trees”, https://github.com/Actyx/banyan
[107] Horn, Petr Jan. “Autonomic Computing: IBM's Perspective on the State of Information Technology.” (2001).

Funded by the European Union