TaRDIS

D6.2: Report on the second iteration of TaRDIS toolbox components

Includes references to and descriptions of software prototypes.

Revision: v.0.1

Work package	WP6
Task	T6.1, T6.2, and T6.3
Due date	31/12/2024
Submission date	10/01/2025
Deliverable lead	NOVA
Version	0.1
Authors	João Leitão (NOVA), Nuno Preguiça (NOVA), Miloš Simić (UNS), Diogo Jesus (NOVA), João Brilha (NOVA), João Bordalo (NOVA), Dimitra Tsigkari (TID), Tomás Galvão (NOVA), Rafael Matos (NOVA), Felipe Carmos (NOVA), André Rijo (NOVA), Carla Ferreira (NOVA), João Gonçalo Pereira (NOVA), Diogo Paulico (NOVA), Rafael Costa (NOVA), Tamara Ranković (UNS)
Reviewers	??
Abstract	This deliverable presents the progress of the TaRDIS project consortium on the design of novel programming abstractions for Swarm systems, distributed abstractions for communication and membership primitives, distributed data

Grant Agreement No.: 101093006
Call: HORIZON-CL4-2022-DATA-01
Topic: HORIZON-CL4-2022-DATA-01-03
Type of action: HORIZON- RIA

	management systems, and monitoring and self-management of decentralised systems. The report further presents and discusses software prototypes and demos that were developed in the context of WP6 during the second year of the project. We also discuss how the different components of the TaRDIS toolbox developed in the context of this work package can be leveraged by different types of Swarm systems and applications, and further provide indications of how these tools (and other currently under development) will be integrated across different TaRDIS use cases and demos.
Keywords	Swarm Programming Abstractions, Decentralised Membership Protocols, Decentralised Communication Protocols, Distributed Management Systems, Decentralised Monitoring, Decentralised Management, Decentralized Machine Learning

Document Revision History

Version	Date	Description of change	List of contributor(s)
V0.1	26/12/2024	Defined structure of the deliverable	João Leitão (NOVA)
V1.0	04/04/2025	Complete draft of deliverable	João Leitão (NOVA), Nuno Preguiça (NOVA), Miloš Simić (UNS), Diogo Jesus (NOVA), João Brilha (NOVA)
V1.1		Revision and content added	

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 developments of Work Package 6 (WP6) of the TaRDIS project until the end of the second year of the project. WP6 is responsible for researching, designing, and implementing the fundamental building blocks and runtime infrastructure that support the execution and management of swarm applications, and as such, contributes substantially to the ongoing development of the TaRDIS toolbox. The current deliverable documents the continuation of this work by presenting both conceptual advancements and software artefacts developed over the reporting period.

Activities in WP6 are organized into three interrelated tasks. Task 6.1 focuses on providing decentralised membership and communication primitives. Building on the efforts and abstractions reported in Deliverable D6.1, the team developed and evaluated additional protocols and services, including variants of the HyParView protocol, a probabilistic global membership service, gossip-based and flood-based broadcast protocols, and an anti-entropy mechanism for state reconciliation. Furthermore, multiple implementations of these protocols were added to the Babel ecosystem and extended with support for mobile platforms and autonomic behaviors through Babel-Android and Babel-Swarm, respectively. These developments improve the composability and adaptability of communication services in the context of dynamic and heterogeneous swarm environments.

In Task 6.2, which targets decentralised data management and replication, efforts continued in enhancing and integrating solutions that address the challenges of data consistency and availability across the cloud-edge continuum. The Arboreal system was further validated and enriched as a hierarchical replication scheme offering causal+ consistency [40,41]. PotionDB was consolidated as a performant, partially replicated, geo-distributed storage layer. Moreover, adapters were developed to integrate third-party systems such as Cassandra, C3, Engage, and Hyperledger Fabric into the Babel ecosystem, allowing developers to leverage existing storage infrastructures within TaRDIS applications. Work has also been carried out to expose these systems through a unified abstraction layer.

Task 6.3 made significant progress in supporting the monitoring and reconfiguration of swarm systems. The telemetry acquisition framework was improved with more fine-grained instrumentation across system layers, and new modules were introduced to support metric aggregation and telemetry-driven control. A centralised reconfiguration engine based on hierarchical namespaces was developed to manage components across diverse devices using containerisation. Complementary to this, extensions were made to Babel to allow applications and protocols to expose their own metrics, and control endpoints were developed to facilitate integration with machine learning-based reconfiguration strategies.

These developments also included the integration of IoT devices into the TaRDIS ecosystem, in particular in the context of Babel, demonstrating the applicability of the developed components and innovations of TaRDIS in real-world scenarios involving physical-world interaction, IoT domains, and Domotics, all of which are relevant application

scenarios nowadays. These efforts were developed to increase the reach and impact of the project.

Throughout this iteration, the tools and artefacts produced were validated through demonstrations and experimental evaluations. These include the TaRDIS Messaging App and Voting App, which run on heterogeneous devices and showcase swarm-native communication and coordination. These demos were executed in local lab environments and served both as validation platforms and as illustrative examples of the TaRDIS runtime.

The source code of the core tools, protocol implementations, and supporting infrastructure is openly available through the TaRDIS project repositories. This code base includes reference implementations for the proposed abstractions and can be reused or extended in future iterations and other work packages. In the next phase, WP6 will extend its activities to further improve the support for secure communication and data sharing, expand the use of decentralised intelligence for autonomous control, and continue coalescing components into production-ready building blocks for the TaRDIS toolbox.

TABLE OF CONTENTS

1. INTRODUCTION 9
2. PROGRESS REPORT And PLAN 10
a. Overview 10
b. Programming Abstractions for Swarm Systems 10
i. The Babel Framework (background) 11
Requirements for Enhancing Babel for Swarm Systems 11
ii. Babel Ecosystem: Babel-Swarm 12
Motivation 12
Design and Implementation 13
Self-Configuration 13
Enhanced Security 13
Adaptive Management 13
Evaluation 14
Experimental Setup 14
Resource Consumption 15
iii. Babel Ecosystem: Babel for Android 16
Network Management in Android 17
Java Runtime and Common Libraries 17
Android Application Life-Cycle and Persistent Connections 17
iv. Babel API for Web Services 19
v. Babel 2: Evolving Babel Ecosystem for Further Swarm Domains 20
vi. Generalizing the Babel Approach 21
c. Decentralised Membership and Communication Primitives (T6.1) 23
i. Evolving Membership Abstractions for Self-Configuration, Self-Management, and Security 24
HyParVIew with Self-Discovery 24
HyParView with Autonomic Management 25
HyParView with Security 27
ii. Membership Abstractions to model Satellite Swarms 28
iii. Membership Abstractions for (hierarchical) Decentralized Communities 29
d. Decentralised Data Management and Replication (T6.2) 30
i. PotionDB: Eventual Consistent Materialized Views and Distributed Query Processing 31
Overview 32
Data model 32
Interface 33
Consistency 34
Architecture 34
Transaction and replication protocols 35
Objects 35
Transaction processing 36
Replication 37

Views 37
Generated objects and triggers 37
Developer-defined views 38
Status 39
ii. Extensible CRDT Library for the Babel Ecosystem 39
iii. Nimbus Decentralized Storage 40
iv. Exploring Decentralised Solutions by Byzantine Settings 46
e. Decentralised Monitoring and Reconfiguration (T6.3) 47
i. Docker Monitorization and Telemetry Acquisition 48
ii. Metric Aggregation 49
iii. Metric and Telemetry APIs for Centralized Machine Learning 50
iv. Epidemic Dissemination of Telemetry 50
v. Building Swarm Models through Machine Learning 51
vi. Enabling Decentralized Machine Learning with TaRDIS Toolbox 52
f. Other Integrations and Activities 54
i. IoT Device Integration 54
3. A GUIDE TO THE TARDIS TOOLBOX WP6 COMPONENTS 57
a. Membership Abstractions 57
b. Communication Primitives 60
c. Data Management Solutions 63
d. Monitoring Solutions 65
Applications 66
Tardis Simple Usecase 66
Decentralized Voting Application 67
Others 67
4. USE CASE PROGRESS REPORT 69
a. Telefónica 69
b. GMV 69
c. EDP 70
d. Actyx 70
5. SOFTWARE 71
a. Overview 71
b. Babel Ecosystem 71
c. Reconfiguration and Monitorization tool based on Namespaces 72
6. DEMOS 74
A. TaRDIS Messaging APP 74
B. TaRDIS Voting APP 76
7. STATE OF THE ART REVISION 80
a. Frameworks for Developing Decentralized and Swarm Systems 80
b. Decentralised Membership and Communication Primitives 80
c. Distributed Data Management System 81
d. Decentralised Monitoring 82
e. Self-Management of Distributed Systems 83
f. Decentralized Machine Learning 83

8. PUBLICATION AND DISSEMINATION ACTIVITIES **85**
a. Publications 85
b. Dissemination Activities 85
9. RELATIONSHIP WITH OTHER TECHNICAL WORK PACKAGES **86**
a. WP3 86
b. WP4 86
c. WP5 86
d. WP7 86
e. WP8 87
10. CONCLUSIONS **88**

1. INTRODUCTION

This document reports on the progress and results achieved in Work Package 6 (WP6) of the TaRDIS project during the second year of the project. WP6 is responsible for the research and development of the runtime systems and low-level abstractions that support the deployment and execution of decentralised applications in dynamic and heterogeneous environments. These building blocks constitute the foundation of the TaRDIS toolbox and provide core capabilities such as decentralised communication, data sharing, reconfiguration, and monitoring.

The activities carried out during this period build upon the design principles, architectures, and components introduced in the first project year and documented in Deliverable D6.1. In particular, the work has focused on extending the runtime with additional protocols, consolidating core abstractions, and evolving the Babel framework and its ecosystem into a more expressive, modular, and portable foundation for swarm-based application development. This includes the development of autonomic runtime capabilities (Babel-Swarm), support for mobile and Android-based platforms (Babel-Android), and the integration of tools for self-observation, adaptive control, and telemetry-driven reconfiguration.

The deliverable also documents on the relevant progress in the development of decentralised data management and storage solutions, including the maturation of PotionDB and Arboreal, the integration of third-party systems through uniform interfaces, and the continued refinement of abstractions for partially replicated and eventually consistent data access. In parallel, new mechanisms were designed to improve the monitoring and management of distributed deployments, including fine-grained telemetry acquisition, metric aggregation, and the centralised reconfiguration of application components via hierarchical namespaces.

This document is structured as follows. Section 2 provides an extensive report on past activities and future plans in the context of the different tasks and also covering activities that are cross-cutting these tasks. This section starts with an in-depth description of the Babel framework and its two major extensions that were developed in the context of TaRDIS (Babel-Swarm and Babel-Android). Section 3 provides short descriptions for the several tools of the TaRDIS toolbox that were developed in the context of WP6 . Section 4 discusses the ongoing activities of WP6 for the use cases of TaRDIS. Section 5 describes the main software artefacts and tools developed and made available through the project repositories. Section 6 details the demonstrators implemented during the reporting period. Section 7 presents a revision of the relevant state of the art, identifying the key gaps addressed by the work conducted, while Section 8 reports on the publications and dissemination activities of this period. Section 9 summarizes the main activities being carried out in collaboration with other work packages. Finally, Section 10 finishes this document.

Funded by the European Union

2. PROGRESS REPORT AND PLAN

a. OVERVIEW

During the second year of the TaRDIS project, WP6 has made substantial progress in advancing the runtime and systems support for decentralized and adaptive swarm applications. A major focus of this period was the consolidation and evolution of the Babel framework, including the development of Babel-Swarm, which introduces autonomic capabilities such as runtime adaptation, monitoring, and protocol composition; and Babel-Android, which extends the runtime to Android platforms, enabling mobile and heterogeneous devices to participate in swarm applications.

Building on the protocol abstractions introduced in the first year, WP6 has developed and validated a wide range of membership and communication protocols—including new variants of HyParView, the X-BOT optimization overlay, epidemic global membership, gossip-based and flood-based broadcast, anti-entropy reconciliation, and random tour-based size estimation—now available as modular and composable components in the Babel ecosystem.

In the area of decentralized data management (Task 6.2), significant strides were made in the maturation of PotionDB and Arboreal, offering partial replication and causal consistency across the edge-cloud continuum. The main achievement of this task was however a fully decentralized data management solution leveraging Conflict-free Replicated Data Types (CRDTs), and built on top of Babel-Swarm, named Nimbus.

In Task 6.3, WP6 has delivered an extensible telemetry acquisition and aggregation framework, extended the Babel runtime with metric-exposing capabilities, and implemented a centralised reconfiguration engine based on namespaces. These tools enable the runtime monitoring, control, and adaptive deployment of swarm applications across distributed infrastructures.

Throughout this period, the developed artefacts were validated and demonstrated through multi-device experiments and two fully operational swarm applications: the TaRDIS Voting App and the TaRDIS Messaging App. These served as both technical validation and illustrative use cases for the integration of protocols, storage, reconfiguration, and UI-less execution across diverse platforms.

Main Planned Activities
In the final phase of the project, WP6 will focus on strengthening the integration and robustness of the components developed so far, and on evolving the runtime to better support autonomic and intelligent behavior. Specific attention will be given to:

- Finalizing and validating the design of Babel 2, the unified and modular successor to the current Babel variants, aligned with the GFDS abstraction.
- Extending support for secure communication and configuration in swarm applications, using simplified and composable cryptographic abstractions.
- Advancing the integration of decentralized machine learning protocols, including dynamic delegation, coordination for split/federated learning, and runtime monitoring.

- Expanding the monitoring and reconfiguration toolset, including tighter integration with machine learning systems for decision-making.

- Increasing the coverage and maturity of demonstration applications, with additional deployments in realistic, distributed environments, including Android-based and mobile swarms.

b. Programming Abstractions for Swarm Systems

Developing and deploying swarm systems, characterized by their decentralized and dynamic nature, can easily become a daunting task for software developers. These systems often require intricate coordination among multiple distributed protocols, each addressing specific functionalities and providing different guarantees and abstractions for the swarm application, including but not limited to, communication, data consistency, and fault tolerance. This inherent complexity arises from managing interactions across different distributed-protocols, handling concurrency, ensuring system robustness, and optimizing performance under varying operational conditions, which involves many times addressing low-level aspects in the development that are both error-prone and distracts the developer from the core functionalities and correctness of those abstractions.

A dedicated framework tailored for swarm systems development is crucial to address these challenges effectively. Such a framework should abstract low-level implementation complexities while providing modular and reusable components that allow developers to focus on higher-level logic. This is particularly important for ensuring that protocols in swarm systems can adapt dynamically to changes in topology, recover from partial failures, and maintain operational efficiency in resource-constrained environments.

Moreover, the rise of user-interactive applications in distributed systems emphasizes the need to extend support to mobile platforms, such as Android. This is mostly motivated because mobile devices increasingly serve as interfaces for users to interact with these systems, providing a unified development framework that encompasses such devices can significantly enhance usability and adoption. By reducing the entry barrier for developers and facilitating the creation of performant and dependable swarm applications, a specialized framework can accelerate innovation in this domain.

Motivated by some of these aspects, and also by the time-consuming frequent development of distributed systems and protocols prototypes in the context of both research, but also in educational settings - particularly in master's courses - at NOVA, and prior to the start of the TaRDIS project, we developed the Babel framework. Babel has been quite successful at supporting both research and pedagogical activities at NOVA, and hence it became a natural basis for the development of powerful and reusable abstractions to support the operation of swarm systems in the context of TaRDIS. Hence on TaRDIS we have been evolving Babel, to better address frequent challenges found on highly heterogeneous and dynamic swarm systems.

In the following, we start by - for both clarification and self-containment - discussing the design of the original Babel Framework [2] prior to the start of TaRDIS (Section B.B.i), then we discuss in more detail the evolutions of Babel that have been conducted in the context of TaRDIS, that have generated different variants of Babel to which we call Babel Ecosystem (Section B.B.ii, Section B.B.iii, and Section B.B.iv), and finally we discuss plans for evolving and unifying this ecosystem to be carried in the final year of TaRDIS (Section B.B.v).

i. The Babel Framework (background)

The Babel framework [2], as originally conceived, addressed critical pressure points for users during the development of distributed protocols. Babel's design emphasized simplicity and performance, enabling developers to focus on the core logic of protocols without being encumbered by the intricacies of low-level operations. Key features of the framework included:

Event-Driven Model: Babel promoted an event-driven programming paradigm, wherein protocols were modeled as state machines reacting to events such as timers, network messages, or intra-process notifications. This approach streamlined the implementation of distributed algorithms by abstracting away concurrency and execution management complexities.

Networking Abstractions: Babel introduced extensible networking channels, providing developers with the flexibility to implement a wide range of communication patterns, from peer-to-peer interactions to client-server models. The framework supported seamless integration of custom channels, ensuring adaptability to diverse application requirements.

Protocol Modularity: Developers could design protocols independently, leveraging Babel's core to manage inter-protocol interactions and ensure consistent message delivery across processes. This modularity facilitated the reuse of existing implementations and reduced development overhead.

Java-Based Implementation: By leveraging Java, Babel combined strong typing with a robust object-oriented approach, providing developers with a familiar and efficient environment for building distributed systems.

Requirements for Enhancing Babel for Swarm Systems

To address the unique demands of swarm systems, Babel must evolve beyond its initial design. Key requirements for the next iteration of the framework include:

Security: Comprehensive support for security mechanisms, including protocol authentication, encrypted communications, and access control, to safeguard interactions in potentially adversarial environments.

Self-Configuration: Automated mechanisms for the dynamic configuration of protocols and applications, enabling seamless adaptation to changes in topology, resource availability, or environmental conditions.

Self-Management: Capabilities for autonomously managing protocol lifecycles, monitoring performance, and optimizing resource allocation to ensure system robustness and efficiency.

Android Support: Extension of Babel's functionality to Android devices, recognizing the critical role of mobile platforms as user interfaces in swarm applications. This enhancement will enable the development of interactive, user-centric distributed systems that leverage the ubiquity of smartphones and tablets.

ii. Babel Ecosystem: Babel-Swarm

The increasing complexity of decentralized systems demands robust frameworks to assist developers in implementing protocols that are not only performant but also adaptable, secure, and capable of autonomous management. To address these challenges, we present Babel-Swarm, an evolution of the original Babel framework. Babel-Swarm extends Babel's capabilities to cater to modern swarm systems, integrating mechanisms for self-configuration, enhanced security, and adaptive protocol management, all seamlessly incorporated into the Babel-Core.

Motivation

The development of decentralized applications has become increasingly challenging due to the dynamic nature of such systems, coupled with their need for high performance, robust security, and autonomous operation. Key motivations for the development of Babel-Swarm include:

Dynamic Membership Management: Decentralized systems often require nodes to dynamically join and leave the network. The lack of predefined configurations or mechanisms for automatic discovery creates significant barriers.

Security Requirements: Ensuring secure communication and trust among nodes in the absence of centralized authorities is a critical challenge.

Adaptive Behavior: Applications must adapt their operational parameters dynamically to accommodate changes in environmental conditions, such as fluctuations in node count, workload, or available resources.

Babel-Swarm addresses these challenges, empowering developers to build more robust and feature-rich decentralized applications while reducing development complexity.

Design and Implementation

Babel-Swarm introduces significant enhancements to the Babel-Core, integrating new abstractions and functionalities to support self-configuration, security, and adaptive management. These features are implemented as modular extensions, ensuring compatibility with existing Babel applications while enabling advanced capabilities.

Self-Configuration

Babel-Swarm simplifies the onboarding of nodes into decentralized systems through two key mechanisms:

Node Discovery: The Babel-Core includes a new DiscoveryProtocol abstraction that supports broadcast and multicast techniques for locating peers. Developers can extend this abstraction to implement custom discovery protocols, which are dynamically loaded during runtime using reflection.

Dynamic Parameter Management: Protocol parameters can now be annotated with @AutoConfigureParameter. The Babel-Core introspects these annotations during initialization, automatically fetching values from existing nodes or external sources like DNS records. Two primary implementations—parameter replication from active nodes and DNS-based parameterization—are provided as built-in options.

Enhanced Security

Recognizing the critical need for security in decentralized systems, Babel-Swarm incorporates:

Identity Management: Babel-Swarm supports self-signed certificates for node authentication. While this does not fully authenticate the node, it allows to entwine different interactions with a node over time, which is beneficial in some contexts. The framework's identity manager provides APIs for handling cryptographic material, enabling developers to implement customized trust models.

Secure Communication Channels: A new SecureChannel abstraction ensures that all communications are encrypted and authenticated. Each connection undergoes an initial handshake where certificates are exchanged and validated against customizable trust policies.

Runtime Cryptographic Operations: The Babel-Core leverages Java's Cryptography Architecture, with extensions for runtime cryptographic operations using libraries like Bouncy Castle. This includes key generation, digital signatures, and encryption/decryption primitives.

Funded by the European Union

Adaptive Management

To enable autonomous operation, Babel-Swarm integrates:

Adaptive Parameters: Developers can annotate protocol parameters with @Adaptive, allowing these values to be adjusted at runtime based on environmental metrics.

Autonomic Controller: The Babel-Core now includes an autonomic controller that monitors system metrics (e.g., network size, latency) and triggers parameter reconfiguration as needed. For example, a gossip protocol's fanout can be dynamically adjusted to optimize message dissemination.

Metrics Collection and Estimation: A lightweight RandomTourProtocol is provided to estimate network size, with results feeding into the autonomic controller for informed decision-making.

These features were integrated into the Babel-Core with minimal overhead, leveraging Java's reflection mechanisms to dynamically manage parameter annotations and configuration changes.

Evaluation

To validate Babel-Swarm's capabilities, we conducted an extensive evaluation using a test application that integrates three protocols: a membership protocol (HyParView), a gossip-based dissemination protocol, and an anti-entropy protocol for reliable message delivery.

Experimental Setup

The evaluation was conducted using a cluster of servers with high-performance specifications to ensure reliable and scalable testing. The cluster consisted of two configurations: one featuring AMD EPYC 9124 processors with 256 GiB of RAM and dual 10Gbps network interfaces, and another using Intel Xeon Gold processors with 128 GiB of RAM and similar dual 10Gbps network interfaces (operating in bound mode). These machines provided a robust environment for simulating a variety of distributed scenarios.

Docker containers were deployed to simulate a network with varying node counts (25, 50, 100). Each node executed the test application, transmitting messages at regular intervals. The protocols were instrumented to measure key metrics, including:

Latency: Time between message transmission and delivery.

Reliability: Fraction of messages successfully delivered.

Resource Usage: CPU and memory consumption.

Below we discuss the results for resource consumption, since in terms of latency and reliability the introduction of these new features have not exhibited noticeable differences.

Resource Consumption

Plots below illustrate the Memory and CPU consumption of the test application under different configurations:

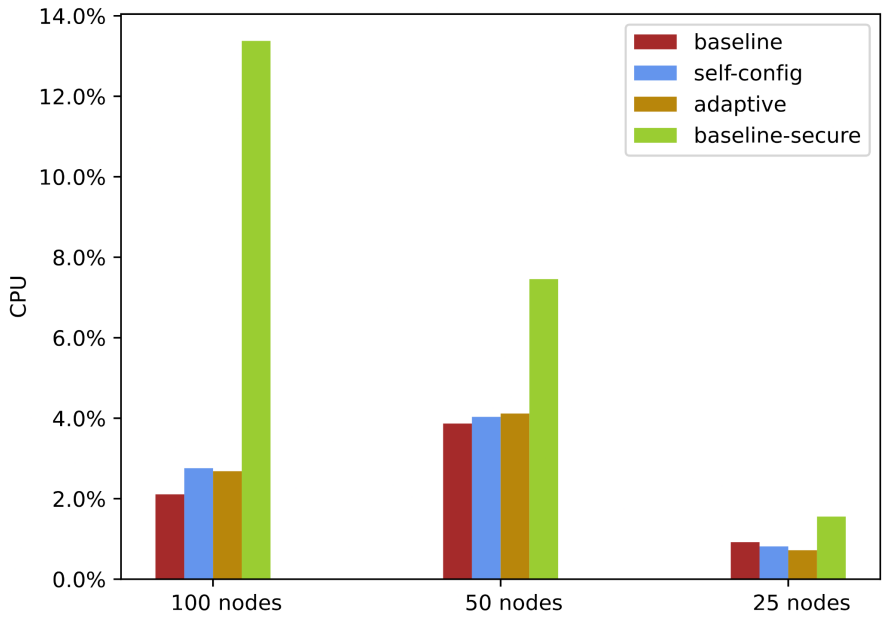


Fig: CPU consumption of Messaging Application using different features of Babel-Swarm

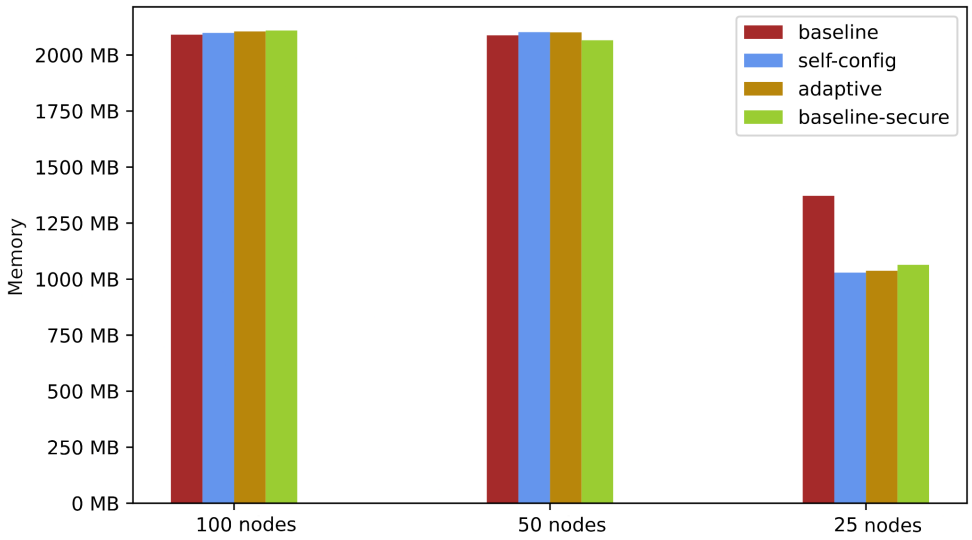


Fig: Memory consumption of Messaging Application using different features of Babel-Swarm

Baseline vs. Secure: The secure variant exhibited slightly higher CPU consumption due to cryptographic overhead but did not exhibit a noticeable overhead in terms of memory across all network sizes (being better for 25 nodes).

Self-Config and Adaptive: Both variants achieved comparable CPU and memory consumption to the baseline, demonstrating that Babel-Swarm's advanced features do not compromise pero.

iii.Babel Ecosystem: Babel for Android

Mobile phones are inherently relevant to swarm applications, since they can provide an accessible and simple interface for users to interact with the swarm as a whole (both for monitoring and management) [16]. Due to this we have decided to port the Babel-Swarm framework (and some of the distributed protocols developed for it) to the Android ecosystem.

Such an effort is reasonable since the Android ecosystem also leverages Java (and Kotlin, an idiom of Java) to develop applications. There were however three main challenges that had to be addressed in porting Babel-Swarm to the Android environment:

1) Interacting with the network in the Android OS requires interacting with the Network Manager service of Android.
2) Java runtime and common libraries had to be addressed, in particular the Android ecosystem only supports Java up to version 17, whereas Babel-Swarm was developed using Java version 22 to be able to take advantage of the new features (and performance) of the language.
3) The application life-cycle in Android is controlled by the operating system, and applications that are not in the foreground can be put in a paused state. This is a challenge because some of the protocols operating in Babel require keeping communication channels (e.g., TCP connections) open to other devices, and such connections can be dropped when the application is put into pause.

We have taken a pragmatic approach to deal with these challenges that allowed us to have swarm applications developed using Babel-Swarm running on Android devices (we have performed preliminary evaluations on several types of devices including different models of mobile phones and tablets). In the following we briefly discussed how we addressed the aforementioned challenges.

Network Management in Android

In the current prototype of Babel for Android, and to explicitly deal with the operation of the Network Manager, we had to modify the logic of the (internal) Discovery protocols used by Babel to identify other processes in the local network that are running the same application,

which internally is then fed to decentralized membership protocols as a contact node (i.e., an entry point for a new node to join). The modification was required to associate multicast/local area broadcast mechanisms to a particular network interface of the device that is both active, supports these protocols, and has a valid IP address at the time when the babel application starts.

In the future we plan to support network changes, namely changes of IP address or even the interface (i.e., WiFi vs 4G/5G) that provides connectivity to the device over the lifetime of the application.

Java Runtime and Common Libraries

Due to the fact that Android only supports Java 17, we had to create a fork for both the core of Babel-Swarm and all decentralized protocols that we wanted to experiment within the Android ecosystem and port them from Java 21 to Java 17. This required removing some Java 21 functionalities that were not present in Java 17. Moreover, the Log4J2 logging library that we were using on Babel-Swarm is not effective in the Android ecosystem, therefore we did remove calls to Log4J2 from the code (it should be noted that these Log entries were mostly used for debugging purposes).

Android Application Life-Cycle and Persistent Connections

Since Babel maintains active connections with other members of the swarm, it is not enough for an Android application that takes advantage of Babel to use it as a library. This is because the Android operating system manages the lifecycle of the application, and as such, when the application is not in foreground, or when the screen is locked, the operating system might put the application in pause, which leads to the termination of TCP connections due to inactivity. Additionally, protocols that require periodic management operations, for instance membership management protocols, become unable to perform those actions, leading those devices to become disconnected from the entire swarm.

To deal with this aspect of the application life-cycle management we have encapsulated the Babel runtime and support protocols (specific for an application) in a bundle that runs as a foreground service. This allows the Babel runtime to keep operating even when the application is not in foreground or when the phone is locked. To achieve this, we have the Babel-Android core and Communication layer as a Java library that can be imported within the scope of a template for a Foreground service that loads all decentralized protocols required for a specific application. This foreground service provides APIs for the Android application to both interact with it (i.e., make requests to specific protocols) and for those protocols to provide asynchronous notifications to the application. Given this development model, the Android application itself is only responsible for managing the interface with the user, exposing information and collecting inputs.

To simplify this interaction, the programmer should extend the Application class of the Android runtime to initialize the foreground service when the application starts and disable it when the application terminates. Furthermore, this extension of the Application class can materialize the APIs to allow the application (interface) logic to interact with the Babel runtime and its associated distributed protocols. We have conducted preliminary

assessment of the overhead (in terms of energy consumption) of this approach, by running seven different Android devices running the TaRDIS Messaging application together with four Raspberry Pi 4, running the linux version of this application, where each application instance on a Raspberry generates 64Kb random messages every 30 seconds with a probability of 10%. All nodes (Android devices and Raspberries) were inter-connected by an instance of the HyParView protocol [17][1] using a single gossip-based broadcast[2] strategy to disseminate messages collaboratively, and a simple anti-entropy protocol[3] to recover missed messages, while also running a distributed protocol to allow each individual node to compute a local estimate of the size of the network[4] named Random-Tour [18]. This simple evaluation showed that all Android devices could operate in this ecosystem, receiving all messages, for at least 48h before their battery was exhausted, which seems acceptable in this context.

iv.Babel API for Web Services

When working with multiple programming languages, developers often encounter significant challenges in making them interact seamlessly. Each programming language is designed with its own syntax, rules, and paradigms, reflecting a unique approach to solving problems. Moreover, different languages often operate in distinct execution environments. Some languages are interpreted, like Python or JavaScript, while others are compiled, like C or Go. These differences affect performance, memory usage, and how data is processed. When combining these languages in a single application, developers must consider how to synchronize execution models, which can lead to inefficiencies or complications in scaling the application.

Nowadays, most of the applications built for the web are developed using some kind of script language (e.g., Java Script, Python, etc.) and utilize external APIs (e.g., REST) to communicate with other services, such as a back end providing a database.

With this in mind, the Babel API for Web Services offers a generic framework to enable web applications and services (i.e., written in other programming languages) to interact with the Babel environment being executed in the Java Virtual Machine.
This is possible by interfacing Babel protocols with REST APIs and Web Sockets, such in a way that when a client wishes to interact with the Babel environment, it executes a HTTP request (or sends a message through a previously setup Web Socket) to issue an operation. When this operation is finished in Babel, a response is sent back to the client.

[1] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/protocols/membership/hyparview-with-discovery
[2] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/protocols/communication/eagergossipbroadcast
[3] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/protocols/communication/antientropy
[4] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/protocols/network-estimations/random-tour

Additionally, clients can also be notified reactively through the use of Web Sockets when new data arrives through Babel. This way applications can update their UI accordingly as new information flows through each node in the system.

To use the framework, developers should extend their application protocols with the **GenericWebServiceProtocol** class, in order to access the features available to protocols that receive operations through HTTP requests and Web Socket messages. This way, protocols will be notified when new requests arrive, and can then process them internally and respond with the use of callbacks. Moreover, the developer can create their REST resources and Web Socket intances by merely extending the **GenericREST** and **GenericWebSocket** classes respectively, and passing the **GenericWebServiceProtocol** in charge of handling the requests. An interaction of this flow can be seen in the figure depicted below:

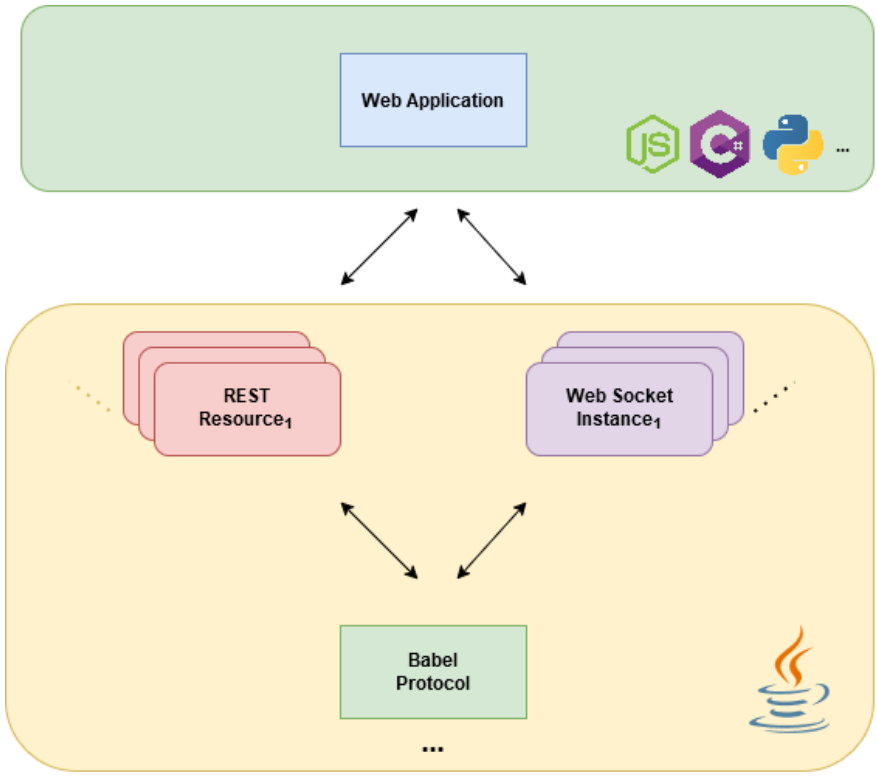


Fig: Babel Web API Architecture

Repository

More details of the framework, functionalities and the proper documentation can be found in Babel API for Web Services - Core[5], and a few examples are available under Babel API for Web Services - Examples[6].

[5]https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-webservices/babel-webservices-core
[6]https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-webservices/web-services-examples

v.Babel 2: Evolving Babel Ecosystem for Further Swarm Domains

Babel 2 represents our current plans and ongoing efforts to build a de-facto implementation of the Generic Framework for Dynamic Decentralized Systems (GFDS) (see Generalizing the Babel Approach), translating the framework's concepts into a cohesive, concrete platform for building resilient and adaptive swarm applications. While GFDS is conceived as a language-agnostic architectural blueprint, Babel 2 will remain a Java-only implementation for the foreseeable future. This decision is guided by the need to consolidate the existing Babel variants—Babel-Core, Babel-Swarm, Babel-Android—into a unified codebase built on Java 17, which ensures broader compatibility across desktop, server, and mobile environments.

The primary goal of Babel 2 is not only to evolve the current codebase but to systematically address the recurring challenges observed across the development and deployment of decentralized systems using Babel. Among these, the integration of security mechanisms has traditionally posed a significant barrier. While Babel-Swarm introduced support for encrypted channels and certificate-based authentication, these features required intricate configurations and exposed a fragmented API surface. Babel 2 will streamline these capabilities through more intuitive abstractions, providing developers with secure-by-default communication primitives that are both flexible and simple to use.

Another critical direction in Babel 2 is the unification and simplification of communication abstractions. Existing Babel-based protocols often had to explicitly bind to a particular transport technology—such as TCP, UDP, or WebSockets—making them cumbersome to reuse across different execution contexts. Babel 2 introduces a redesigned communication layer where transport protocols are treated as fully interchangeable modules. Protocol developers can now write logic that is entirely agnostic to the underlying transport, enabling seamless deployment across heterogeneous networks, from ad-hoc local networks to wide-area cloud deployments.

Additionally, Babel 2 integrates more robust self-configuration and runtime adaptability features by default, building upon the reflective configuration capabilities of Babel-Swarm. Support for Android remains a key concern, and adopting Java 17 as the base version ensures compatibility with the mobile ecosystem, while retaining access to modern language features that support clean, maintainable code.

In practical terms, Babel 2 is more than just a next version of the framework—it is a convergence point. By subsuming the existing Babel variants into a unified runtime and harmonizing the various protocol implementations, Babel 2 enables reuse, extensibility, and maintainability at a new scale. It becomes the backbone not only for continued TaRDIS developments but also for any future work based on GFDS, serving as a stable, extensible reference that other implementations—potentially in other languages—can eventually follow.

As the project enters its final year, the consolidation and validation of Babel 2 will be a key focus. The goal is to deliver a mature, tested, and integrated platform that embodies the principles of GFDS while remaining grounded in the practical requirements of real-world

swarm applications. This effort will not only benefit TaRDIS use cases but also establish Babel 2 as a robust foundation for future innovation in the design and deployment of dynamic decentralized systems.

vi. Generalizing the Babel Approach

As mentioned previously, Babel, and more precisely its new version Babel-Swarm and the upcoming release Babel 2, aim to assist developers in implementing decentralized dynamic systems by offering a plethora of features to build and manage distributed applications at a large scale. While Babel is designed to be flexible and modular, and continuous to evolve in order to meet up with the ever increasing demand of swarm systems, at the moment, its reference implementation is solely in Java. Thus, in order to allow other developers to benefit from the functionalities offered in Babel and its simple but powerful design, we propose a *Generic Framework For Building Dynamic Decentralized Systems (GFDS)*. GFDS generalizes the Babel approach by leveraging the already existing abstractions presented in Babel and expanding on them:

Workgroup:	Network Working Group
Internet-Draft:	draft-jesus-gfds-latest
Published:	February 2025
Intended Status:	Standards Track
Expires:	29 August 2025
Authors:	D. Jesus J. Leitão
	TaRDIS TaRDIS

A Generic Framework for Building Dynamic Decentralized Systems (GFDS)

Abstract

Building and managing highly dynamic and heterogeneous decentralized systems can prove to be quite challenging due to the great complexity and scale of such environments. This document specifies a Generic Framework for Building Dynamic Decentralized Systems (GFDS), which composes a reference architecture and execution model for developing and managing these systems, while providing high-level abstractions to users.

GFDS aims to offer a set of tools, abstractions, and best practices to allow developers to design, deploy, and manage distributed applications that are dynamic, resilient, scalable, and fault-tolerant. The framework is composed of an execution model, architecture and

extended examples, which details all of the relevant aspects to write and build a reference implementation.

Due to its nature, GFDS is completely language agnostic and encompasses the fundamental concepts and components to build a framework for building decentralized systems. This design choice encourages interoperability and enables developers interested in this framework to implement their own version.

The document is being written as a recently submitted internet-draft [7] (i.e., a working document submitted to the Internet Engineering Task Force (IETF) for discussion and potential standardization) . The working version of the draft is available in the TaRDIS repository [8].

c. Decentralised Membership and Communication Primitives (T6.1)

The mission of Task 6.1 is to provide fundamental abstractions for decentralised membership and communication that are essential to the operation and coordination of swarm systems. These systems, which typically operate in open and heterogeneous environments without centralized control, demand mechanisms that enable nodes to discover each other, maintain knowledge of the system's composition, and exchange information efficiently and reliably.

In this context, Task 6.1 addresses two core technical challenges: first, how to model and implement membership services that allow each node in a swarm to be aware—completely or partially—of the other participating nodes; and second, how to build communication primitives that enable rich interaction models such as point-to-point messaging, gossip-based dissemination, publish/subscribe, and broadcast, while remaining agnostic to underlying transport mechanisms.

This task builds upon extensive experience and state of the art in peer-to-peer and overlay network protocols, leveraging their strengths while addressing key limitations such as rigidity in API design, poor interoperability, and a lack of composability. The approach taken in TaRDIS focuses on creating generic, reusable APIs for membership and communication services, enabling application components to be decoupled from specific protocol implementations. This abstraction layer empowers developers to swap out underlying protocols to adapt to different environments or operational requirements without changing application logic.

The ongoing work in this task continues to push forward in refining these abstractions, incorporating additional protocols, and improving their usability and efficiency. In the next phase, particular emphasis is being placed on simplifying developer experience, integrating intuitive security mechanisms, and abstracting away transport-level concerns to support more dynamic, robust, and interoperable swarm applications.

[7] https://datatracker.ietf.org/doc/draft-jesus-gfds/
[8] https://codelab.fct.unl.pt/di/research/tardis/wp6/drafts/gfds-internet-draft

In summary, this task develops and validates:

- Underlying abstractions for supporting the development and efficient operation of higher-level data management (T6.2) and reconfiguration services (T6.3).
- 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 two years of the TaRDIS project:

i. Evolving Membership Abstractions for Self-Configuration, Self-Management, and Security

The Babel-Swarm variant of Babel (Described above in Babel Ecosystem: Babel-Swarm) created the opportunity to evolve different membership protocols to take advantage of the new features introduced by this variant. To showcase and evaluate these features, as well as motivated by the fact that HyParView [16] is the membership abstraction that was more flexible in terms of scalability and providing resilience to swarms infrastructures, we have created different variants of this protocol exploiting these features.

HyParVIew with Self-Discovery

This version of HyParView was redesigned to take advantage of the ability provided by Babel-Swarm to allow a Swarm membership protocol, when a new node tries to join the swarm, to automatically discover another member of that swarm that can serve as an introduction node (sometimes called a contact node in the literature) to facilitate the new node to join the membership abstraction (ensuring that both the new node becomes aware of some other nodes in the system, and that some other nodes in the system become aware of the existence of the new node).

The current abstraction operates by taking advantage of pluggable Discovery mechanisms on Babel-Swarm. In particular, we have developed two such mechanisms, one based on IP-Multicast, another on local network Broadcast (hence these mechanisms are only useful in the context of local networks, we do plan to generalize this by also considering

alternatives such as DNS name resolution or the creation of the registry service that can be leveraged to support this mechanism).

The implementation was straightforward and easy to achieve using the mechanisms provided by Babel-Swarm. The new variant of the protocol extends the Discoverable Protocol (instead of the GenericProto of the original babel) that notifies the runtime that this protocol supports self-discovery. If no contact node is provided in the configuration, the Babel-Swarm runtime is notified, and the plugable discovery protocol uses its mechanisms to find another node in the network belonging to the same swarm, and upon finding it notifies this version of the protocol. At this point the protocol can initialize and join the overlay network of HyParView, and the node becomes itself able to serve as contact node to other nodes that can join the system. Due to the minimal overhead introduced by this mechanism, and the fact that this mechanism allows for simpler configurations on applications that do not depend on knowing a-priori the address of a member of the swarm, we currently use this as the de-facto implementation of this membership abstraction. This version was also, due to these reasons, ported to the Babel-Android variant.

The implementations of this variants (for each version of Babel can be found in:

- https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/membership/hyparview-with-discovery
- https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/protocols/membership/hyparview-with-discovery

HyParView with Autonomic Management

This new version of HyParView actually combines two new features introduced by Babel-Swarm. First it allows a node to be started without being aware of the configuration parameters in this protocol (in particular, HyParView features a total of nine parameters that control its behaviour as discussed on [16]). We call this Self-Configuration. The second feature is that this variant of the protocol allows for (some of) these parameters to be modified at runtime to improve the operation of the protocol when the operational conditions change. We achieve this in practice in the current prototype by taking advantage of a distributed protocol that allows each node to produce a local estimate of the network size, and an Autonomic Controller component that exists (independently) at each node that takes into account the current estimate of the membership size to manipulate the size of the partial views maintained by HyParView[9]. We call this Self-Management.

The mechanism that allows for the self-configuration takes advantage of a plugable special protocol on the Babel-Swarm core. In particular, we have developed a solution based on TXT records on DNS addresses. This allows to ship nodes where the configuration only reports what is the DNS name that should be looked-up to extract the (current) ideal

[9] We note that the theory of epidemics points towards an ideal size of the active view of HyParView to be ln(N) + 1, where N is the total number of nodes in the system. There are however some restrictions, for instance, it is very unlikely that a random K-graph will be connected for values of K below four. Moreover, the fact that active views in HyParView are sysmetric, allows - considering the restriction identified above - to use log(N) + 1 instead of ln(N) +1 when computing the the ideal number of neighbors (i.e., the active view size) of each node.

starting configuration for a node. Again the implementation of this feature is straightforward using the mechanisms provided by Babel-Swarm. It was sufficient to have the protocol implementation extending the AdaptiveMembershipProtocol class instead of the GenericProto class of the original Babel, define a new parameter to have the DNS name where the configuration should be looked up on bootstrap, and avoid the initialization of parameters with default values. This is enough for the runtime of Babel-Swarm to identify this protocol as supporting self-configuration, looking up the configuration on the DNS service, and setup the appropriate parameters on the protocol before the protocol starts its operations (we should note that this feature is compatible with the self-discovery mechanism reported above, also the contact node can be provided - in a somewhat less dynamic way - as an DNS TXT entry as well).

The second feature, self-management, as previously mentioned requires additional complexity. In terms of protocol adaptation this is also straightforward, as the previously mentioned AdaptiveMembershipProtocol abstract class already features support for this aspect of the protocol. However, this feature does require the programmer to specify special setter and getter methods for all protocol parameters that can be changed at runtime. The setters in particular require the programmer to verify if the configuration change is valid, as this aspect is intimately entwined with the protocol logic.

In addition to these changes to the protocol, we require two other components, one that extracts information at runtime that can inform or guide the runtime reconfiguration of a protocol, and one local component that takes this information, analysis it, takes decisions regarding changes to the current configuration, and issues commands to the protocol using a standard management API (that is materialized by the previously mentioned AdaptiveMembershipProtocol abstract class). To study the viability of our design and validate it we have explored several fully decentralized approaches to infer, at runtime, the size of the swarm. After evaluating a few different alternatives, we decided to use an implementation of the Random-Tour protocol, which operates by having each node periodically sampling the network using random-walks. To avoid incorrect individual estimates to create unnecessary perturbations on the system, our Autonomic Controller operates using a window of a configurable number of samples to make decisions. The simple controller that we implemented averages the 10 different and consecutive samples and, when the system size grows or decreases too much it issues reconfiguration commands to respectively increase or decrease the number of (active and passive) neighbors maintained by HyParView. Preliminary evaluation has shown that this approach yields adequate results, allowing nodes to modify their configuration while avoiding oscillatory behaviours that could easily have a negative impact on the overall operation of the Swarm.

We have also started to apply these core concepts to gossip-based communication abstractions, as to adjust the fanout employed by the protocols based on the network size estimate. An initial version of this protocol is already implemented and is currently being evaluated.

The components discussed in this section have their code available at the following locations:

- https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/membership/hyparview-autonomic
- https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/network-estimations/random-tour
- https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/autonomic-controllers/simple-autonomic-controller
- https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/communication/adaptive-eager-gossip-broadcast

HyParView with Security

Finally, we have implemented a variant of HyParView that provides some fundamental security mechanisms for a distributed environment. In particular, this version of HyParView allows each element of the swarm to be identified by a certificate, which allows nodes, when establishing communication channels with other members of the swarm, to validate that their peer is a valid participant, by automatically receiving its certificate and validating it (using a policy defined by the developer, as to easily allow for enforcing different restrictions, such as the certificate being signed by a specific set of entities, or by other (trusted) swarm nodes). In addition to this, messages exchanged by the protocol are also signed, which allows for messages to be used to prove that some node has exhibited some misbehaviour (i.e., signatures provide the property of non-repudiation over messages sent by a node). Finally, communication channels are encrypted to provide both integrity and privacy of information exchanged among nodes.

These functionalities, at the Babel-Swarm level, are provided by a combination of a Security manager in Babel-Swarm, that handled identify (cryptographic material) management and verification, as well as message signing and verification, combined with transparent mechanisms provided by a new Secure Communication Channel developed specifically for enriching Babel-Swarm, that for instance can verify the validity of peer identities transparently. Encrypted communication is also provided by the Secure Communication Channel abstraction.

In terms of protocol changes, this is the feature of Babel-Swarm that requires more effort from the programmer, although this is mostly related with correctly setting up the Secure Communication Channel, which includes interactions with the Identity abstraction provided by the security manager.

We have also created a variant of the Gossip-based broadcast protocol and our simple anti-entropy protocol that incorporates the available security mechanisms. This was mostly motivated because the secure communication abstractions (in terms of providing the properties of non-repudiation, integrity, and privacy) are more plausible at the level of communication abstractions, since these are the distributed protocols that effectively transfer application data, that can with a higher probability, benefit from having the aforementioned properties.

We have however noted during the development of these variants of the protocols that the API developed to expose security mechanisms in Babel-Swarm was too complex, in the sense that there are multiple mechanisms to achieve its functionality, mostly motivated to provide flexibility to a wide range of application scenarios. Due to this, we are currently revising these security abstractions in the context of the development of Babel version 2, to expose them to the programmer in a simpler and more intuitive way.

The interested reader can find the code for the protocol variants discussed here in:

- https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/membership/secure-hyparview
- https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/communication/eagergossipbroadcast-secure
- https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/communication/anti-entropy-secure

ii. Membership Abstractions to model Satellite Swarms

Motivated by the challenges related with managing swarms of satellites, or more generally, swarms of mobile devices whose communication patterns evolve over time depending on the relative positions of these devices to each other and potentially other factors (e.g, this phenomena can also happen on swarms of drones or even in vehicular networks), and to support the demonstrator of TaRDIS for the use case of partner GMV, we have identified the need of a different type of decentralized membership management abstraction, that can capture and deal with this dynamic nature of opportunity for two swarm elements to communicate and exchange information. This is fundamentally different from other membership abstractions that have been explored in the context of WP6, that usually assume the existence of a communication subtract (e.g., IP network) that offers, while devices are active, a mechanisms for them to communicate, that despite being best effort is most of the times functional.

We have started to work on this new type of membership abstraction that fundamentally takes an opportunistic approach to identify opportunities for swarm devices to interact with each other, either based on probing the communication medium (for instance a wireless medium), or by taking advantage of a known a-priori, but potentially imprecise, information about the relative positions of nodes and a scheduling of possible interactions between them.

We will have a prototype of this solution, also integrated into Babel-Swarm (or most likely Babel version 2) that will be used to develop the demonstration of the GMV use case, where we will model the communication scheduling between nodes, based on a virtual model of a swarm of satellites, empowering the protocol to be able to autonomously readjusts when the predicted position of nodes is not met due to external factors.

It should be noted that this future contribution of WP6, will allow to run realistic simulations of the GMV use case using real code, that can be at a later date be ported to other

platforms to support its integration in a real use case scenario. We plan to fully report on the results of this activity in Deliverable D6.3.

iii. Membership Abstractions for (hierarchical) Decentralized Communities

Also motivated by the challenges related with the construction of a decentralized and reliable renewable energy market from partner EDP, WP6 is currently working on the development of membership abstractions with special characteristics to allow for the implementation of the demonstrator of this use case. The implementations, will be carried out also using Babel-Swarm (and later ported to Babel version 2 when this is finished) and will be therefore able to execute in a myriad of devices (as long as they can execute the Java virtual machine (JVM) paving the way for a the development of a real product to be exploited by the partner in the future.

The EDP use case is a paradigmatic example of a swarm that benefits from biasing the interaction among swarm elements to those that are in close physical proximity. This is related to the fact that transferring energy is faster and more efficient across small distances. Therefore, a decentralized renewable energy community will benefit the most from the interactions among close participants, although in some extreme cases it might be useful to be able to contact more distant participants, or even the core of the system, to avoid situations where excessive energy production can generate overloads. To support this we need a membership abstraction that automatically organizes nodes participants in a logical network (i.e., an overlay network) that on one hand promoted neighboring relationships between nodes in close proximity, while at the same time having some form of hierarchical topology that can be leveraged to iteratively look for participants that need to acquire or provide energy. Achieving this design in a robust and efficient way without resorting to a centralized control point is challenging.

Due to the inherent challenges related with this activity we currently have two lines of ongoing work to build a membership abstraction for hierarchical decentralized swarms that depart from different base solutions to try to achieve and effective solution that can be employed to support the use case of EDP and other applications that have similar requirements.

The first line of work exploits a technique named Biasing of overlay networks [19], that in a nutshell departs from a purely random overlay, and iteratively allows participants in that overlay to explore the system and coordinate to exchange existing links by better links, in this particular case, swapping overlay links between distant nodes by links among closer nodes. This work is therefore departing from the X-BOT protocol [20,21] that does this in a coordinated manner, having groups of 4 nodes iteratively swapping two links by two better overlay links, while keeping a configurable number of distant links unchanged as to ensure the global connectivity of the system, and adapting it to achieve two complementary goals: 1) ensure that nodes maintain a large number of neighbors that are in their physical vicinity, allowing the emergence of communities by the natural operation of the protocol; and 2) manipulate the distant (i.e., unbiased) links to allow the emergence of a hierarchy among nodes, that can be leveraged to effectively coordinate exchanges among nodes

through requests and offers, in the particular case of the EDP use case, of renewable energy.

The second line of work tries to achieve an abstraction with similar properties departing from a different point, in particular, it is being developed departing from an existing overlay management solution named Overnesia [22], that is a fully decentralized membership protocols that builds an overlay network where nodes are organized in cliques at random. These cliques are then interconnected (also at random) to ensure global connectivity. In this line of work we are changing this approach, that already forms specific cliques of nodes that can be easily mapped to a (closed) community in our use case, to ensure that these cliques take into account the physical proximity of swarm elements, ensuring that nodes in close vicinity are within the same clique. Moreover, we plan to take advantage of explicit information about the geographical location of nodes in each clique to interconnect these cliques not at random, but using a logic that allows multiple independent and location-aware hierarchies to emerge. Such a solution will allow us to explore a design for a decentralized renewable energy market that can explicitly define operations over the local community, neighboring communities, and the global system.

Both of these activities are still ongoing and are planned to end within the next three months. It should be noted that the decision to pursue independent lines of research to tackle this challenge is related with the risk associated with this activity, as there is no previous work that provides similar properties as the ones required by this use case. Final findings related to this activity will be reported on Deliverable D6.3 and we expect to integrate one of these solutions into the prototype of the EDP Use case.

d. Decentralised Data Management and Replication (T6.2)

Task 6.2 focuses on providing the necessary abstractions and protocols for decentralized data management and replication to support the development and operation of dynamic and heterogeneous swarm applications in TaRDIS. In such applications, nodes are highly diverse in terms of resources and capabilities, and may include powerful cloud servers, lightweight edge devices, and mobile clients with intermittent connectivity. Supporting these systems requires flexible and efficient data management mechanisms that can adapt to the changing conditions of the swarm and ensure that relevant data is consistently available where it is needed.

To address this, Task 6.2 is designing and developing solutions that go beyond conventional full replication or static consistency models. A key challenge is the need to support partial replication, where each node holds only a subset of the data and can dynamically update its interests over time. This requires algorithms capable of reconfiguring replication strategies as nodes move, join or leave, and as their data access patterns change. In parallel, the task must ensure that such replication maintains acceptable levels of consistency for applications, particularly through mechanisms that avoid excessive coordination while ensuring convergence.

Another significant aspect of Task 6.2 is supporting heterogeneous operating conditions. This includes enabling data access and update even when connectivity is disrupted, and providing strategies to merge concurrent updates in an eventually consistent manner using Conflict-free Replicated Data Types (CRDTs). Furthermore, to improve the efficiency of data processing and dissemination, the task is exploring the use of materialized views for common queries, allowing nodes to locally maintain and access precomputed results of interest without querying the entire swarm.

While current work has focused primarily on these mechanisms, future developments will extend the system with support for Byzantine fault tolerance, ensuring the system remains correct even in the presence of misbehaving or compromised nodes — an increasingly relevant concern for decentralized applications operating in open environments.

In summary, this task develops and validates:

- Replication mechanisms that support heterogeneous settings, including nodes with limited computational and storage resources and intermittent connectivity.
- Partial replication algorithms that adapt to changing access patterns and mobility, ensuring nodes store and retrieve only the relevant subset of data.
- Efficient support for recurrent queries using materialized views maintained incrementally and in a consistent fashion.
- A set of CRDT-based abstractions, including a novel extensible CRDT library for the Babel ecosystem.
- A fully decentralized storage layer designed for dynamic and intelligent swarm applications.
- Initial foundations for supporting Byzantine fault tolerance in decentralized data management systems.

During the second 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 3.D.i presents the evolution of PotionDB, focusing on supporting recurrent queries over geo-distributed data.
● Section 3.D.ii presents the CRDT library designed for the Babel framework.
● Section 3.D.iii presents Nimbus, a fully decentralized storage system designed for supporting intelligent swarm applications.

i. PotionDB: Eventual Consistent Materialized Views and Distributed Query Processing

PotionDB is a geo-distributed database, in which each replica only holds a subset of the database. In this context, performing queries is challenging, as no single replica has all the necessary data to answer the query. In D6.1, we presented the protocols to perform transactions that access data present in different replicas. Executing such transactions incurs an important overhead, as remote replicas need to be accessed. For recurrent

queries, which are common in applications, an alternative approach would be to maintain materialized views with the result of such queries.

In this deliverable, we focus on the mechanism to maintain materialized views for supporting recurrent queries. Implementing such features efficiently in a partially geo-distributed database requires addressing two main challenges. First, it is necessary to maintain the materialized views consistent with the remaining data accessed by the application. To achieve this, we build on the replication protocols presented in D6.1, extending them to support view objects. Second, it is necessary to efficiently support materialized views. Our key insight is that most updates do not affect the state of views for common recurrent queries with aggregations or limits (e.g. TPC-H queries). We have designed an incremental view maintenance mechanism that only propagates the necessary updates, which builds and extends Non-uniform Conflict-free Replicated Data Types (NuCRDTs) [3].

Overview

PotionDB is a geo-distributed database.PotionDB adopts partial geo-replication, with data items replicated only at some locations. This allows PotionDB to reduce replication cost when compared to full replication, saving on both storage, processing, and networking costs. While some application operations access data objects directly, other operations require data that results from an aggregation. For supporting the latter, PotionDB provides materialized views that are the result of an aggregation over geo-partitioned data and provides algorithms to efficiently maintain these views consistent.

Data model

PotionDB is a distributed key-value database. We identify two types of values: base objects, Objs, and derived objects, Views. The set of objects of a database is defined as DB = Objs ∪ Views. Informally, an object, o, is any value that is either a base object, b, or view, v. We distinguish between base objects and views whenever necessary for clarity.
The value of a view is a function over the values of other object(s), i.e., $\forall v \in$Views : v = fun$v(Dv)$, with Dv the set of base and derived objects used to compute v. We assume the computation is non-recursive, i.e., the value of a view is never based on its own value or, formally, $\forall v \in$Views : $v \notin Dv$.
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, maps and top-K objects. Both base object and views are implemented as CRDTs [4], guaranteeing that object replicas converge to a single state in the presence of concurrent updates.

Interface

PotionDB offers a transactional key-value interface, as summarized in Table 3.D.1. 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: (i) get(txId, id), which returns the full state of the object; (ii) read(txId, id,op), which returns the result of read-only operation op executed in the object; and (iii) 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 or rollback(txId) to abort the transaction.

begin(clk) →txId	get(txId, id) →value
commit(txId) →clk	read(txId, id, op) →value
rollback(txId) →ok	upsert(txId, id, op) →ok
oneShotTx(clk, (id, op)+) →clk, value+	

Table 3.D.1. PotionDB's data manipulation API.

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, and to create views. Even if buckets have no associated data type, we expect that applications store objects of the same type in each bucket. A document/table row can be stored as a map CRDT, with each element of the map having its own type.

```
CREATE VIEW (DailyTopDetects, views) WITH
YEAR = ANY (SELECT DISTINCT SampleDate.Year FROM daydata),
MONTH = ANY (SELECT DISTINCT SampleDate.Month FROM daydata)
DAY = ANY (SELECT DISTINCT SampleDate.Da FROM daydata)  AS
SELECT PollutantName, SampleDate.Day, SampleDate.Month, SampleDate.Year,
MAX(Value) AS Total
FROM daydata
WHERE SampleDate.Day = [DAY] AND  SampleDate.Month = [MONTH] AND SampleDate.Year = [YEAR]
GROUP BY PollutantName
ORDER BY Total DESC
LIMIT 10
```

Figure 3.D.1. Specification of the view Daily TopDetection.

The create view receives a view specification defined in a language based on SQL. Figure 3.D.1 shows the specification of a view that maintains the top 10 detected pollutants in each day. CREATE VIEW specifies the key prefix and bucket of the materialized view objects. The FROM specifies which container(s) are used in the computation of the view, while SELECT specifies the attributes of the view, which can include simple attributes or aggregations. It is possible to define aggregations over groups with the GROUP BY clause, restrict the number of elements with a LIMIT clause, and order the results using an ORDER BY clause.

In recurrent queries, it is common that a generic query is instantiated with different values. To support this, our language allows the use of variables in the view definition. In the example, variables YEAR, MONTH and DAY lead the system to maintain for each date (year, month, day), the top 10 detections.

Given a view definition, PotionDB automatically infers the objects to be used to store the view and the view updates required to incrementally maintain the materialized view whenever a relevant base object is updated. The data in a materialized view object is read as any other object, by issuing reads to the view object.

Consistency

PotionDB is a weakly consistent database that provides Transactional Causal Consistency (TCC) semantics [5]. Intuitively, in TCC 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, the snapshot reflects only updates of committed transactions. Moreover, if a transaction t is included in the snapshot, all transactions that happened-before t are also included. Unlike snapshot isolation, and similarly to parallel snapshot isolation with p-sets [6], two concurrent transactions can modify the same object, with updates being merged using CRDT rules, thus avoiding write-write conflicts. The formal definition was presented in D6.1.

Architecture

We designed PotionDB with partial geo-replication in mind. Thus, we assume PotionDB instances to be spread at different locations. Each location only replicates a subset of the whole data. The system administrator has control over where each object, both base objects and views, is replicated. This allows us to take into account 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.

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. PotionDB has three modules: the Transaction Manager, the Materializer and the Replicator.

The Transaction Manager coordinates transaction execution, executing the TCC protocol detailed in D6.1. The Materializer manages the database objects, which encode the type-specific aspects of PotionDB operation, including rules for conflict-resolution and consistent view maintenance for different object and view types. These rules are encoded in CRDT/NuCRDT objects. The Replicator implements the partial replication protocol

detailed in D6.1. This protocol is type-independent, but it uses information provided by the object in its operation - e.g., when a view is updated, the replicator propagates to other replicas the updates returned by the view object, which can be an empty set if no updates need to be propagated.

Transaction and replication protocols

Transaction and replication protocols were already introduced in D6.1. In this deliverable, we present the change introduced to support views.

Objects

PotionDB stores two types of CRDTs: common CRDTs [4] and non-uniform CRDTs (NuCRDTs) [3], which encode the type-specific aspects of PotionDB operation.

CRDTs. 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 serialization must respect the happens-before relation, thus guaranteeing that PotionDB replicas converge.

Our prototype supports the following CRDTs: last-writer-wins register, for storing opaque values; add-wins set, for sets where adding an elements wins over concurrent removals; add-wins map, for maps of values; counter, for numbers that accepts concurrent increments and decrements; average, for maintaining the average of values added to this object; and an integer register, that keeps the max/min value registered.

NuCRDTs. Non-uniform CRDTs [3] are CRDTs that guarantee that in a quiescent state, the observable state of all replicas is the same. Two observable states are defined as equivalent iff, for each possible read operation, the result is equal when executed on either state.
Unlike normal CRDTs, in NuCRDTs, during the replication process, it is only necessary to propagate updates that may affect the observable state. For example, consider a maximum object with the insert(n) and getMax() operations. An insert executed in a replica only needs to be propagated to other replicas if the inserted value can be the new maximum.

NuCRDTs allow saving on both replication, processing and storage costs, as not all updates need to be replicated and applied everywhere. In PotionDB, we support the following NuCRDTs: (i) maximum and minimum objects, for storing the maximum or minimum of the objects added to the object; (ii) top-K, for storing the K entries with largest values; (iii) top-K counter, for storing the K (key, value) entries with largest values, where the value can be updated by issuing increment/decrement operations. NuCRDTs can be used directly by applications, but are more commonly used for supporting views.

As proposed by Cabrita et. al. [3], by not propagating all operations, in some cases, a NuCRDT may temporarily expose an incorrect state. Consider the maximum NuCRDT. Each replica keeps only the maximum element and the elements that were inserted locally. If the maximum NuCRDT has a remove(n) operation, when the maximum element is removed in some replica, the replica may not have the new maximum, as it may have been inserted in some other replica. All replicas of the maximum NuCRDT eventually converge to the new maximum value, as every replica will propagate the local maximum element after receiving the remove operation, guaranteeing that all replicas will receive the new maximum.

To support views in PotionDB, we had to extend existing NuCRDTs specifications in three ways. First, top-K objects, instead of keeping only the value of the element, keeps multiple attributes for each element, as needed for storing a complete view entry. An application can update the value used for establishing the top elements using a set operation (in top-K) or an increment/decrement (in a top-K counter). Other attributes can also be updated (as in a map CRDT).

Second, we extended NuCRDTs to maintain a larger ob- servable state to reduce the cases in which a replica may be exposing incorrect results - e.g. the maximum NuCRDT maintains the two largest elements, guaranteeing that replicas have the new maximum if a single remove is issued. In most practical situations this guarantees that replicas have the correct values - e.g. a Top-10 object that keeps the largest 20 elements exposes no anomaly unless more than 10 concurrent removes of top elements occur, which is very unlikely in practice.

Third, we extend NuCRDTs to include information to know if it might be exposing incorrect results. This information consists in the timestamps of transactions that could cause the anomaly - e.g. in the maximum NuCRDT, the transactions that remove the maximum. Knowing the updates that replicas have seen (which is maintained by PotionDB), a replica knows that no anomaly can occur if the problematic operation has been seen by all replicas, which would have triggered replicas to send operations relevant for the observable state, if any. PotionDB uses this to block reads.

Transaction processing

The transaction processing algorithm was presented in D6.1. The introduction of NuCRDTs introduces new challenges, as NuCRDTs may temporarily expose incorrect results when an operation op changes the set of relevant operations. PotionDB is able to detect this situation using the summaries of operations - this situation is expected to be rare. Applications may select two behaviors when they start a transaction.First, to ignore potential anomalies and immediately execute the read in the local replica. This guarantees fast replies at the cost of potential anomalies. Second, to strictly enforce TCC. In this case, the read blocks until the replica gathers information that no relevant operation is missing. This requires receiving information from all replicas that op was performed and all new relevant operations, if any, have been received.
This information is propagated in the replication process.

Triggers. PotionDB has an after update trigger mechanism that can be associated with objects in a bucket or container. When a transaction executes at the initial replica, after an update, the trigger runs and it may issue reads and updates to the same or other objects. These updates are committed and propagated to other replicas as part of the transaction. Triggers can be used by applications for any purpose, but its primary goal is to support incremental view maintenance.

Replication

The replication process was presented in D6.1. In this section, we focus on the changes required for the introduction of NuCRDTs. NuCRDTs use a non-uniform replication approach [3], in which some updates might not need to be propagated, as explained in Section 3.1. Not immediately propagating some updates is straightforward, as, when executed locally, an update operation may produce a null effect update if there is no effect in the observable state.

In NuCRDTs, the execution of an operation op may make a previous local operation opp relevant, requiring opp to be propagated to other replicas (e.g. remove(n) in the maximum NuCRDTs makes the insertion of the second maximum element relevant). There are two cases to be considered. First, when op executes in the initial replica, the effect of op will include also the effect of opp. As the combined effects are propagated and applied in the context of the same transaction, no anomaly is generated. Second, when the now relevant opp operation was executed in another replica. This is handled by replication process as follows. When a location receives the replication stream from other replicas, it executes the received effects in the objects' local copy. For NuCRDTs, the execution of an effect operation may generate additional effects - in our example, the execution of the effects of opwould generate the effects of opp. These extra effects are propagated to other replicas along with the information that ophas been executed.

Views

We now discuss how PotionDB supports materialized views.

Generated objects and triggers

We now outline how PotionDB, given a view specification, generates the objects to hold the view's data and its updates.

Generated objects: The objects used for storing the view's data depends on the type of query. If the query has no limit clause and includes either no aggregation or an aggregation of type sum (or similar, such as count or average), maximum or minimum, the full materialized view will be stored in a map CRDT. If the query has a limit clause and no aggregation, or an aggregation of type maximum (or minimum), the top-K is used. If the query has a limit clause and an aggregation of type sum (or similar), the top-K counter is

used. In any case, the elements of the view are maps with multiple columns, one for each attribute specified in the select of the view definition.

For a view that includes variables, multiple objects are used, one for each possible value of the variables. In the top pollutant example of Figure 3.D.1, one top-K object is created (as needed) for each month of each year, and its PotionDB key includes the name of the view plus the month and year. The top-K will have a set of entries, each one with the identifier of a pollutant and the total value. The entry could have additional information, such as the description of the product if that was defined in the view.

Generated triggers: PotionDB generates triggers to update the contents of view objects, as base objects are created, updated or deleted. The triggers will be set for the container (or buckets) specified in the from clause of the view definition. If a where clause is included, updates to objects that do not match the defined condition will be ignored.

As a view may be composed of multiple objects, it is first necessary to determine which view object must be updated. This is achieved from the view definition and the values in the updated base object - in our example, from the date of the measurements, it is immediate to know which view object must be updated. The exact operation generated to update the view object depends on the type of view object, but it consists in applying the same update performed in the base object to the corresponding view entry map. For example, consider the creation of a new measurement for pollutant P, with the value vP. In this case, the top-K of the date of the measurement is updated by incrementing the total value of sales for P by vP. The underlying functionality of top-K guarantees that the updated entries will be propagated to other replicas when necessary, and that replicas keep the correct top-K elements.

Developer-defined views

A developer can create views by defining the objects to maintain the view's data and the triggers to update these view objects. The top-K and top-K counter NuCRDTs include the logic for replicating only the necessary updates to maintain in all replicas the top elements for data that is updated using set value or increment operations. When the view does not consist of the topmost elements, the Map CRDT can be used.

Aggregations defined in the views - e.g. sum, maximum - are supported by using CRDT and NuCRDT objects, such as the counter CRDT and maximum NuCRDT. After defining the objects to be used, the application should define the triggers that (populate and) update the view objects.

Status

PotionDB is under development, and we expect to report the evaluation of the prototype in the next deliverable.

ii. Extensible CRDT Library for the Babel Ecosystem

Synchronizing state in replicated systems is one of the most challenging aspects of distributed computing. In such systems, multiple copies of data are maintained across different nodes to ensure high availability and fault tolerance. However, ensuring that all replicas are consistent and reflect the same state at any given time is difficult due to network latency, failures, and the inherent asynchrony of distributed systems. Different replicas may update concurrently, leading to conflicting changes that must be resolved in a way that guarantees data consistency without compromising system performance. Synchronizing state requires sophisticated protocols, such as consensus algorithms or conflict resolution strategies, to ensure that the system operates correctly even when failures or delays occur, making it a non-trivial task for developers to manage.

CRDTs, or Conflict-Free Replicated Data Types, are a class of data structures designed to resolve the challenges of synchronizing state in distributed, replicated systems without requiring centralized coordination or locking mechanisms. They enable multiple replicas of data to be updated independently and concurrently, ensuring that all replicas will eventually converge to the same state, even in the presence of network partitions or failures. The key feature of CRDTs is that they allow for automatic conflict resolution, ensuring that updates made to different replicas can be safely merged without human intervention or complex algorithms. With this, CRDTs guarantee that no matter in what order or how many times updates are applied across different replicas, after the system stabilizes (i.e., no updates are issued for an amount of time and all replicas receive the same updates) the final state will always be the same). CRDTs are highly relevant for swarm systems, since these systems consist of highly decentralized and independent nodes, designed to operate in environments with intermittent connectivity and limited communication meaning the system must continue functioning even when certain agents or nodes are offline or disconnected from the network. In this context, CRDTs provide a robust way to manage state synchronization across the distributed agents without relying on centralized coordination.

In light of this, TaRDIS developed an extensible and serializable CRDT library for the Babel Ecosystem. This library offers a set of common CRDTs (i.e., Counters, Registers, Sets, Maps, etc.) in the three most common CRDT flavors, operation-based, state-based and delta-based, with the last one containing an highly optimized implementation of CRDTs that support causal-consistency guarantees per object. This library is extensible by allowing developers to implement their own solutions on top of the interfaces provided (and used) among the library, such as depicted in the image below:

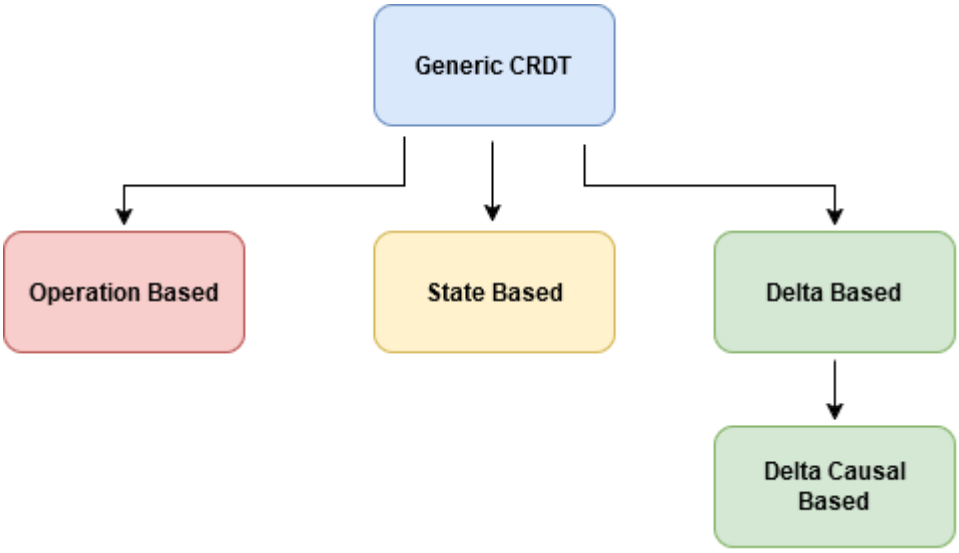


Fig: Babel CRDTs Diagram

If a developer wishes to implement its own solution of a CRDT, he can do it by using one of the interfaces portrayed above and by filling in the common methods that the CRDT of that flavor should possess and by using the common structures (i.e., Version Vectors) offered by the library.

Moreover, this library is used with other tools created for TaDIS, namely Nimbus Decentralized Storage Solution, explained in detail in the next section.

Repository

More details of the library, examples, tests and documentation can be found in Babel CRDT Library[10]. Additionally, we also provide a set of common of APIs for managing the interaction between protocols and CRDTs under Babel CRDTs - Replication Core Commons[11], and a few examples of replication core implementations[12] (e.g., the layer in charge for propagating the different CRDTs among replicas).

iii. Nimbus Decentralized Storage

In highly volatile and dynamic distributed systems, using storage solutions presents significant challenges due to the inherent instability and unpredictability of the environment. Swarm systems, typically consisting of autonomous agents such as drones, robots, or nodes in a distributed network, operate in a decentralized manner, often in environments where connectivity is intermittent, nodes may join or leave unexpectedly, and data needs to be shared across a large number of nodes in real time. These characteristics make

[10] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-crdt/babel-crdts
[11] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-crdt/babel-crdt-replication-core/babel-crdt-rc-commons
[12] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-crdt/babel-crdt-replication-core/rc-implementations

traditional storage models, which rely on stable and centralized systems, ill-suited for swarm applications.

In traditional systems, data is often stored in a central location or in synchronized nodes, ensuring consistency through locks, transactions, or replication. However, in a swarm system, agents may be dispersed over wide geographical areas or may face communication delays, making it challenging to keep all nodes updated in real time. When agents disconnect or encounter communication issues, they may need to store data locally to be able to work offline, but when they reconnect, synchronizing these local updates with other agents becomes a complex task, especially if conflicts arise between concurrent changes. Moreover, the scalability of storage solutions is another crucial issue. Swarm systems are often designed to scale up or down depending on the task or environmental conditions. As the number of nodes increases, the demands on storage also grow, potentially overwhelming the storage network.

While some solutions [1] aim to extend services with peer-to-peer interactions, they still rely on centralized infrastructures to maintain metadata information and connections with the nodes in the network, in our solution we propose a fully decentralized storage solution that doesn't rely on any kind of dedicated infrastructure and can scale up to a large number of nodes.

We note that while TaRDIS has already presented other storage solutions in previous deliverables, namely Arboreal which extends the replication of data towards edge locations dynamically, thus forming a hierarchical structure on the different nodes; and PotionDB which operates across data centres by supporting transactions and enforcing causal consistency, none of which fit into the highly dynamic swarm scenarios presented previously, and thus requiring a more flexible solution.

Overview

Nimbus is designed as a fully decentralized storage system. Nimbus provides scalable and efficient data storage without relying on a central authority. To achieve this, Nimbus doesn't require any kind of dedicated infrastructure (i.e., cloud or edge nodes) such that each node in the system acts as an independent replica and acts as part of the replicated storage system. This aims to target decentralized applications (such as swarms) where replicas may leave or enter the network at any moment and execute operations in any order, without disrupting the correct functioning of the system as a whole. This way, different applications, such as satellite swarms, telemetry swarm systems in harsh environments, to name a few, can interact directly with each other without relying on an external infrastructure to store and synchronize their data.

Nimbus structures its data into *keyspaces* and *collections*, which we will call from here onwards *information units* (explained in detail further ahead). Moreover, since that swarm nodes may be heavily resource restricted, having every node in the system fully replicating every object becomes too expensive, so Nimbus offers a partial-replication mechanism among its different *information units*. This way, each node can specify or request each information it will replicate, without leading to an overflow state in each peer in the system.

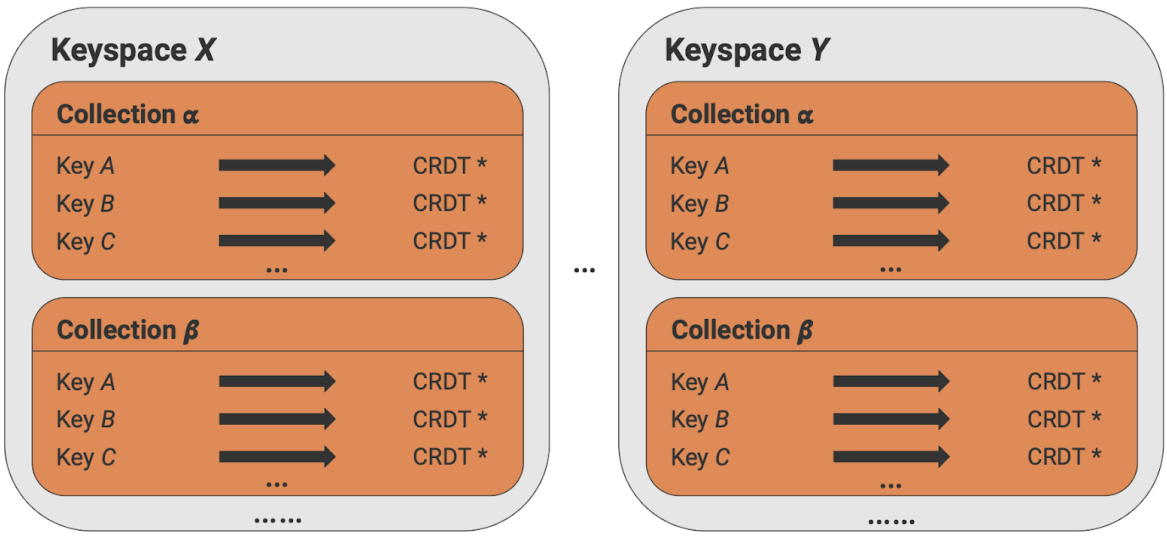
Moreover, in order to tolerate faults, Nimbus supports data persistence by employing mechanisms to persist data on disk on demand or periodically. This way, if a node fails or the system is shut down, a node can load up its state and continue from the point of failure without losing any information.

While swarm nodes act in a highly dynamic way, it is still very important to control the flow of information. Thus, Nimbus supports access control, by allowing the creators of *information units* to set up access policies, and add and remove access in real time. Additionally, it is possible to reconfigure *information units* on demand to better satisfy the swarm needs.

While objects are partial-replicated throughout the different nodes, a *metadata control unit* is shared and propagated to all nodes, so that every node has the information about which *information units* were already created, as well as the access policies for each one and other relevant control information.

Data Model

Nimbus offers a key-value store interface, by having a data model constituted as *keySpaces* and dividing each keySpace into separate *collections,* in other words the *information units*. A collection is represented as a dictionary of key-value pairs, where a key acts as an identifier of an object, and the value is represented as a CRDT (e.g., a counter, set). This offers a rich interface to the developer, by allowing him to choose the data type in which he wishes to encode its data, as well as offering composite types, such as maps, to allow recursive and composite data structures by each application's needs (i.e., a tree-like structure of attributes) as depicted below:



Nimbus Data Model

Interface

Nimbus uses the APIs described in previous TaRDIS deliverables. When a client wishes to interact with Nimbus, he needs only to create a request with the operation he wishes to execute (as detailed in the next table), and when the operation is complete, he will get a proper reply with the information he requested, plus any additional information if something went wrong during the execution (e.g., lack of permissions).
Moreover, Nimbus also supports a notifier reactive mechanism, meaning that when new information arrives from new nodes (i.e., when an update arrives from a neighbor), Nimbus will notify the client so that he can update his application accordingly in real time.

Namely, Nimbus client API follows Babel Protocol Commons,[13] a set of common APIs developed in TaRDIS for interacting with distributed systems and their protocols:

Request: CreateKeySpaceRequest Reply: CreateKeySpaceReply	A create keySpace request is emitted to a node with the given keySpace identifier as well a set of properties for that keySpace. (i.e., permissions). The corresponding reply is sent back with the status of the operation and an optional message.
Request: CreateCollectionRequest Reply: CreateCollectionReply	A create collection request is emitted to a node with the given collection identifier as well a set of properties for that collection. (i.e., permissions). The corresponding reply is sent back with the status of the operation and an optional message.
Request: DeleteKeySpaceRequest Reply: DeleteKeySpaceReply	This operation is issued by a node when it wishes to delete a keySpace from the system. The reply returns the status of the operation and an optional message.
Request: DeleteCollectionRequest Reply: DeleteCollectionReply	This operation is issued by a node when it wishes to delete a collection from the system. The reply returns the status of the operation and an optional message.
Request: ExecuteRequest Reply: ExecuteReply	An execute request is issued by a node when it pretends to interact with one of the objects of the data store. This request encompasses the type of operation (e.g., READ, WRITE, DELETE etc.), the identifier of the object and the corresponding keySpace and collection, as well as the value in the case of being a write operation. The reply contains the

[13] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-protocolcommons

	status of the operation, as well as an optional value in case of being a read operation.
Request: ModifyKeySpaceRequest Reply: ModifyKeySpaceReply	A modify key space request issued by a node when it pretends to reconfigure a given key space. This request allows the change the access control of a keyspace, as well as other relevant metadata information.
Request: ModifyCollectionRequest Reply: ModifCollectionReply	A modify collection request issued by a node when it pretends to reconfigure a given key space. This request allows the change the access control of a collection , as well as other relevant metadata information.
Request: KeySpaceConfigRequest Reply: KeySpaceConfigReply	A request to obtain the configuration parameters of a keyspace. This request is issued when a node wishes to obtain the metadata information of a keyspace (i.e., the access control list).
Request: CollectionConfigRequest Reply: CollectionConfigRequest	A request to obtain the configuration parameters of a collection. This request is issued when a node wishes to obtain the metadata information of a keyspace (i.e., the access control list).
Notification: DataNotification	In order to notify the client of new updates brought up to the node by the background synchronization mechanism of Nimbus, the system issues a notification of the new updates on the objects that the node replicates. This information can be encoded as a JSON object, a plain Java object or a byte array.

Consistency

Nimbus provides strong eventual consistency, a consistency model that guarantees that all nodes reach the same state after receiving all updates, regardless of the order in which updates were applied. This is accomplished by using CRDTs, *Conflict-Free Replicated Data Types,* to synchronize the node's state with the use of epidemic dissemination.

Conflict-free Replicated Data Types (CRDTs) offer several advantages that make them ideal for distributed systems, particularly in environments where consistency and availability are critical, but network partitions are common (such as the case of swarms). This means that even if network partitions occur or some nodes become temporarily unreachable, the system can continue to function normally, and once connectivity is restored, the system will automatically reconcile any differences between replicas.

With this in mind, Nimbus offers a way for nodes to work offline (i.e., due to a network partition of failure) without hindering the user experience. Then it will automatically relay local information to other nodes when the connection is duly established without any intervention by the application.

Replication

As discussed earlier, a key challenge of this work is overcoming limitations and challenges in providing a scalable and reliable storage solution for highly dynamic systems. With this in mind, Nimbus supports partial-replication through its *keyspaces* and *collections*, the *information units*.

Since relying on an autonomic partial replication mechanism can prove to be very expensive and complex due to the highly dynamic nature of this types of systems (e.g., it is very hard to predict when a enter may enter or leave the system), we opted for a mechanism where
creators of *information units* can explicitly alter the replication nodes of each unit. Moreover, if a node has access to a certain *information unit* (e.g., because he requested it previously or was given to by the creator), he can request to be a replica of a given *information unit* from there onwards.

Even if nodes are not replicas of a certain *information unit*, if they have access to it they can explicitly request to access them by contacting another node in the system that replicates such a unit. A depiction of the replication architecture can be seen in the figure below where different nodes, of different natures, replicate different *information units*. It is important to note that the replication is node through two distincts units, the *keyspaces* and *collections*. This way a node may choose to only replicate a certain collection of a keyspace it is only interested in, or the whole keyspace.

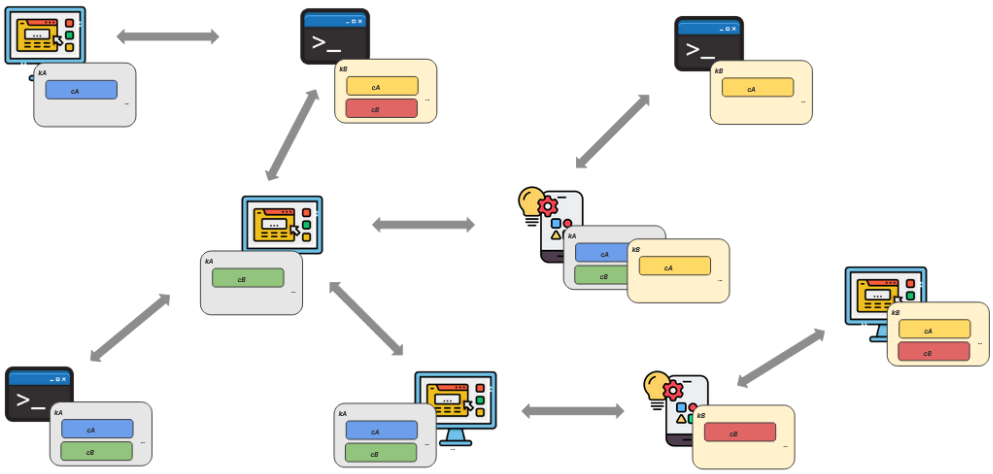


Fig: Nimbus Architecture Overview

This results in a highly scalable solution, where no third party infrastructure is needed to maintain the system data, and where nodes may enter or leave the system without compromising the correctness of the system as a whole.

Status

Nimbus is still under heavy development at the moment of writing this document and going through testing. We expect to have the evaluation of the prototype by the next deliverable, as well as its architecture in full detail.

Repository

More details of the API offered by Nimbus can be found in Babel Protocol Commons[14], under the *storage* package, as well as on the TaRDIS wiki.[15]
An under-development implementation can be found on the Nimbus Git repository.[16]

iv. Exploring Decentralised Solutions by Byzantine Settings

In the context of WP6 we have an ongoing line of work where we are exploring techniques to support partial replication among swarm elements in settings where some of these elements can misbehave. This work is still in its initial steps and is taking advantage of previous work of researchers involved in this project [1].

[14] https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-protocolcommons
[15] https://codelab.fct.unl.pt/di/research/tardis/toolkit/Documentation/-/wikis/TaRDIS-APIs/Data-Management-and-Distribution-Primitives/Decentralised%20Data%20Management%20and%20Replication%20APIs
[16] https://codelab.fct.unl.pt/di/research/tardis/wp6/internal-tools/nimbus

In a nutshell our solution is combining a set of byzantine-fault tolerant decentralized membership abstractions (in particular we are exploring the viability of adapting the Fireflies work [23]) with byzantine-fault tolerant epidemic dissemination mechanisms. We are exploring a data model based on the one provided by Nimbus, where nodes propagate operations among them that are integrated and consolidated using CRDTs.

We plan to report on the findings of this line of work in Deliverable D6.3. We note that none of the Use cases of TaRDIS exhibit strict requirements for Byzantine-fault tolerant decentralized data management solutions, however, this efforts will allow the TaRDIS toolbox to be capable of supporting swarm applications with additional needs.

e. Decentralised Monitoring and Reconfiguration (T6.3)

Task 6.3 focuses on enabling decentralized monitoring and reconfiguration capabilities across the swarm applications and solutions developed within TaRDIS. In highly dynamic and distributed systems, the ability to observe, interpret, and react to runtime conditions is essential to achieving robustness, adaptability, and self-management. Traditional approaches to monitoring and control often rely on centralized components, which introduce bottlenecks, single points of failure, and scalability limitations. In contrast, swarm systems require decentralized strategies where monitoring is distributed and reconfiguration decisions are made locally or collaboratively among nodes.

The goal of this task is to design and implement scalable and composable abstractions for metrics collection, dissemination, aggregation, and interpretation in fully decentralized environments. These abstractions must operate effectively across heterogeneous devices and communication topologies, while preserving the flexibility to integrate with both swarm-native decision-making processes and centralized analytical tools (such as cloud-based machine learning systems).

A critical aspect of the work in this task is enabling runtime reconfiguration of applications and protocols. This involves empowering the system to autonomously tune its behavior—such as adjusting protocol parameters, swapping components, or changing communication strategies—based on observed metrics like latency, resource usage, topology changes, or application-specific KPIs. These reconfiguration actions may be local (e.g., tuning a node's protocol parameters) or global (e.g., triggering collective reorganization). To support this, the monitoring infrastructure must provide not only raw metrics, but also flexible support for defining aggregation functions, thresholds, and triggers that feed into the system's control logic.

The task is also concerned with ensuring that monitoring and reconfiguration mechanisms are lightweight and non-intrusive, minimizing their impact on the performance and energy efficiency of the system—especially on constrained devices. Additionally, they must be resilient to faults and inconsistencies, allowing the system to continue to function and adapt even in the presence of churn, failures, or partial partitions.

In summary, this task develops and validates:

- Decentralized metrics collection mechanisms to monitor the behavior and performance of protocols, applications, and nodes.
- Epidemic and aggregation-based dissemination strategies for metrics sharing and summarization in large-scale settings.
- Interfaces and APIs to enable both local and global reconfiguration based on runtime metrics.
- Lightweight telemetry tools for use in resource-constrained environments such as mobile and edge devices.
- Integration pathways to leverage monitoring data in centralized or offline machine learning systems.
- Support for dynamic adaptation of system behavior based on network size, topology changes, or performance goals.

i. Docker Monitorization and Telemetry Acquisition

Task WP6 provides services that are able to collect, aggregate, transform and store diverse monitoring-related data from heterogeneous sources and with respect to resource availability. The metrics are collected from a few places: (i) node metrics, (ii) applications (running in containers) metrics, and (iii) other places in the toolbox that provide metrics such as distributed protocols and probing mechanisms.

This part of the toolbox leverages the cloud-edge continuum, but it relies on open-source solutions such as *Prometheus*, *Grafana, Docker*, *NodeExporter* as part of its core. Metrics are collected on various nodes and/or applications, and then they are moved to a centralized location and aggregated there. This centralization allows us to open metrics to other interested sides by providing various APIs.

As a second level, we have a collection of application related metrics. For this purpose, we have used *Docker*, an open-source as an industry standard tool for OS-level virtualization of applications. By using *Docker*, we can then rely on its mechanisms to collect and expose various kinds of application-related metrics. Metrics from all containerized applications running inside nodes are then collected, and then transferred to a centralized location for future aggregation and processing. These metrics when combined with node-related metrics gives us more details of what is going on on a single node, and when combined gives us a better picture of what is happening inside running swarms.

To collect node-related metrics, here we consider both hardware and kernel related metrics, and for this task we have used *Node Exporter*. *Node Exporter* exposes a wide variety of hardware and kernel-related metrics. After collecting the node-related metrics, metrics are moved to centralized storage. For this purpose, we used *Prometheus, an open-source* database system with a dimensional data model, flexible query language, efficient time series database and modern alerting approach. Prometheus will pull node-related metrics periodically.

To give a new dimension to the collected metrics, and get even more insight into what is happening in the swarm and/or node(s), we also collect custom metrics, the metrics that are not part of previously mentioned metric types. For example, users may decide to store custom metrics such as protocol related metrics, or tracing information collected by the related tools watching over running applications. These metrics will be stored in the toolbox, and it will be available for utilization by others using exposed APIs. An interface that accepts custom metrics expects inbound metrics data to bi sent in the *OpenMetrics* format.

The system is designed to allow users to define retention periods. These periods are often used to tell the system after how much time, metrics (some of them, or all) can be safely deleted from the storage.

All collected metrics are stored in a centralized place inside a specialized time series database. From this centralized place we can further do two things: (i) visualize metrics changes over time, and (ii) expose stored metrics to others e.g. machine learning systems in the toolbox.

Since we are talking about possibly huge amounts of data, changing over time, it would be good for the people using the service to see changes, and monitor the system from a remote location. For this purpose, the service offers two options: (i) command line interface (CLI), and (ii) visualisation tool. A CLI tool offers basic insight into swarms and nodes. Based on the identifier it shows what are the statues of swarms and nodes at the last aggregation point - when nodes send data to the control plane.

Since the first option does not allow detailed insight inside changes over time, we attached modern visualisation tools capable of handling such complex scenarios regarding data. For this purpose, we used industry standard tools. *Grafana* is a multi-platform open source analytics and interactive visualization interface. It can produce charts, graphs, and alerts for the web when connected to supported data sources.

ii. Metric Aggregation

As we rely on the *OpenMetrics* format to both transfer and store metrics-related data, we operate within constraints provided by its data model. Depending on the specific measurement, we have counters, gauges and histograms as metric types. We use labels for recording the relevant metadata. Node and swarm identifiers are always set, no matter if the metric is related to the node, container or an application. For container metrics, a unique container identifier label is also set, as well as an application name label for custom application metrics. We filter the desired metrics by these labels. Labels represent arbitrary key value pairs. Both key and value are of text type. This simple technique allows us to swap storage engines if needed, since most modern time series engines allow labels in their data model. Even if users do not want to use time series engines, labels are easily added to any storage engine.

Swarm-level metrics are generated on demand by aggregating available node-level metrics. For the most part, calculations are done by summing or averaging values of measurements from individual nodes in that swarm, e.g the total disk usage or the percentage of used memory. However, more advanced processing is required for some metrics, like the swarm CPU utilization. Based on the number of cores per node and CPU time spent in each mode, we obtain the final value by subtracting the total rate of all cores spent in idle mode from the total number of cores in the cluster.

After aggregation and when metrics are stored, the control plane can offer these metrics to other interested sides using publicly available APIs. Other sides e.g. other parts of the toolbox or other users of the system can use these metrics for their distinct needs.

iii. Metric and Telemetry APIs for Centralized Machine Learning

The TaRDIS toolbox will provide primitives to export fine-grained stored metrics to other parts of the toolbox, such as machine learning components that require time series of the stored metrics over some period of time. Provided primitives export fine-grained stored metrics using the standard *OpenMetrics* format. A unified interface of the system and application state progression over time is accessible to users. This will allow for informed decision making.

The toolbox offers APIs for machine learning models to be both trained on and put in the role of decision makers, which provides the automation of the decision process. The telemetry data is being stored by the toolbox in time series manner (i.e. data points with time). The telemetry data means, for instance, CPU load, disk usage, network properties… and their changes and usage over time.

This data is accessible to ML models through a specifically designed APIs directed towards their needs. An agent that requires data from the platform for training, simply needs to send a point in time when the last contact occurred. As a result, the agent will receive a collection of metrics that will be returned in *OpenMetrics* format. If the agent for whatever reason misses to get data in e.g., regular intervals, there is an interface to get metrics in a specific period between two points in time. This ensures that machine learning models can be trained at their own pace. Even if an agent crashes, or goes offline for some extended period of time, newly added data points cannot be missed or skipped.

iv. Epidemic Dissemination of Telemetry

Current implementation of metrics collection and aggregation relies on centralised strategies. This strategy emphasises cloud-edge continuum, and it relies on open-source solutions but has its limitations. Nodes inside swarms might not always have sufficient resources to transfer big amounts of data to some centralised solution. To tackle this issue, as part of the future work we are planning to extend the system with epidemic dissemination of the telemetry information.

Nodes inside the swarm already rely on various communication protocols that allow them to know who their neighbours are (membership protocols e.g. HyParView), or get missing information (e.g. Anti-entropy). This is done via gossip protocols. We can use the existing communication protocols and already opened channels to disseminate telemetry information from the swarm. To further reduce data that needs to be sent to centralized storage, we can do metrics aggregation inside the swarm.

Using gossip-style communication between nodes in the swarm we can collect metrics from all nodes, and do the aggregation of the metrics on the spot, and only broadcast the aggregated information to the control plane (Epidemic broadcast trees e.g. Plumtree). Here, we will explore different kinds of data compression techniques to further reduce the amount of data sent. Since we are collecting time series data, we can apply many techniques that are standard in this field (e.g. delta, delta-delta, run length encoding compression etc.).

Using this peer to peer metrics aggregation and transfer to control plane, we must be cautious about losing valuable data. This is important to the machine learning models, which requires variety for their training. To preserve valuable information for these models, we will explore dynamic techniques in which metrics need to be preserved and/or aggregated in specific ways so that we do not lose important points in time.

Information collected in this way also has one benefit. In the centralized setup, nodes are sending metrics related data to the control plane, and control planes use that data to determine are nodes running or not (alive or dead). This aggregated approach removes the burden from both nodes and control planes from potentially treating these messages as denial of service attacks. If we send combined information to the control plane, from that message we can distinguish which nodes are down, and which are alive and well.

v. Building Swarm Models through Machine Learning

In highly decentralized systems with numerous heterogeneous nodes, such as in the swarm model, managing the system is a complex task, which can be daunting for humans and be error prone. The difficulty emerges due to the large number of components that potentially form the swarm system as well as the number of external factors that can affect both its correctness and performance.

Systems which are capable of self-healing and self-optimizing, are called autonomic systems. Most autonomic systems operate based on static rules and heuristics (usually produced by a domain expert that has a high level view of the consequences of different parameters over the operation of the system), which can become inadequate as the system evolves and environmental conditions change. Furthermore, creating models that capture these types of systems is unfeasible, due to their high complexity.

To autonomically manage swarms, in the context of WP6 and in collaboration with researchers from Telefónica in WP5, we are exploring the alternative of leveraging on

machine learning models for both learning how to manage and perform management actions over the system. As far as we know, there are no available datasets for these types of systems, and as such we have created an emulated environment that replicates the complexity of a real-world deployment of a swarm system. Furthermore, this environment allows for introducing realistic changes over the system at runtime, such as new nodes joining the system, increasing the message transmission rates, and nodes crashing or departing the system during the execution. These changes are reflected in the health metrics of the system, such as its average latency, reliability, and redundant message rate. Using this tool, we have started to train machine learning models that can learn how to automatically manage the system, based on the real-time observation of aforementioned metrics. After this step we will be able to design autonomic managers that based on these models can ensure their correct, and real-time adaptation, to evolving conditions.

This work is focusing currently on the simple TaRDIS Message Application, as this is a simple use case, that combines multiple components developed to support general swarms (such as decentralized membership and communication primitives developed in the context of WP6), and as such provides an adequate case study for these efforts. We plan to report on the final results of these efforts on D6.3 and will study the viability of applying this solution to one of the use cases in the TaRDIS project.

vi. Enabling Decentralized Machine Learning with TaRDIS Toolbox

One of the main goals of TaRDIS is to bring the power of decentralized machine learning mechanisms to swarm systems. These efforts have been conducted in the context of WP5 with different frameworks being proposed to support machine learning model training in different distributed settings and environments. One of these tools has been integrated with the Babel Framework, taking advantage of its membership and communication abstractions to support the interaction between multiple nodes. However, supporting fully decentralised machine learning processes in a highly heterogeneous and dynamic environment might require additional specialized support that falls on the scope of the activities of WP6.

To overcome this limitation in the state of the art, WP6 has therefore started a research activity focused on addressing decentralized machine learning within the TaRDIS project, that is focused on the design and development of a fully decentralized, self-managed collaborative learning platform that seamlessly integrates Federated Learning (FL) and Split Learning (SL) paradigms. These efforts aim at addressing limitations of traditional FL systems, particularly their reliance on central aggregation servers, which introduces inherent risks in terms of privacy, scalability, and fault tolerance. By leveraging the TaRDIS toolbox, and in particular the Babel framework, this initiative aims to deliver a solution that distributes both computational load and control logic, enabling dynamic role assignment, robustness to churn, and intrinsic support for data privacy.

We are building a system where participating devices are capable of configuring and reconfiguring themselves in real time, depending on the changing availability of resources and network conditions. This adaptability is essential to achieve meaningful

decentralization, not only in terms of data locality but also in terms of execution autonomy. A critical aspect of this involves the definition of novel delegation protocols, which allow devices experiencing computational overload to identify and delegate parts of the training task to peers that are idle or underutilized. These protocols must operate efficiently, without introducing excessive overhead, and must preserve the (currently limited) privacy guarantees inherent to SL and FL.

At the time of writing we already built a working prototype that runs on heterogeneous devices, including Raspberry Pi units, utilizing WLAN and LAN connectivity to facilitate communication. Babel-Swarm has been leveraged to manage the underlying peer discovery and perform message exchange, while serving as the backbone for the orchestration of training sessions. In its current form, the system supports centralized FL, using an integration with standard AI tools written in python, using local standard I/O channels between Python-based training modules and the Babel coordination layer. This architecture already enables self-configuration of participants and automatic neighborhood formation, providing a solid foundation for the shift toward full decentralization.

We are currently evolving this solution to support a hybrid FL/SL training model, in which devices may simultaneously assume multiple roles—including client, helper, or aggregator—based on runtime conditions. The delegation mechanism, currently under design, will be taken advantage of to encapsulate telemetry-driven decision-making, allowing nodes to assess peer suitability for task sharing based on factors such as computational capacity, bandwidth availability, and current workload. This dynamic role allocation will be made possible through new Babel-based abstractions specifically designed to manage SL training cycles, including cut-layer coordination and activation exchange.

To support broader platform diversity, particularly the inclusion of Android devices (motivated by the Telefonica Use Case), the ongoing development will transition the system to Babel 2 as soon as it becomes available, which will allow it to explore the possibility for integrating FLaaS and PyTorch in this solution.

We will report on the final achievements of this line of work in Deliverable D6.3, and we plan to explore the viability of its application to the Telefónica Use Case within the scope of activities of TaRDIS.

f. Other Integrations and Activities

This section reports on the complementary goals of WP6, which is related to integration with other tools and ecosystems. During the reported period, and to strategically demonstrate the benefits of the technology and solutions being developed by TaRDIS to the domain of IoT (and more generally domotics) we have decided to explore the integration of control and data acquisition over IoT devices in the context of the Babel ecosystem. These efforts were materialized in the form of a significant extension of an existing open-source library called GrovePi and a set of Babel protocols that take

advantage of this library to expose an event-driven API to other Babel protocols and applications, that we now discuss.

i. IoT Device Integration

Overview

The integration consists of a collection of Babel protocols that aim to provide a uniform and standardized interface for controlling and monitoring physical devices such as sensors and actuators.

These protocols are built on top of an extension — in practice, a complete overhaul — of an existing Java library that is essentially a portion of the larger project that is GrovePi. This overhaul consists of the integration of new I^2C and digital devices into the library, making use of the underlying Pi4j library to allow for interoperability between Raspberry Pi models.

Use Case

This allows users of the Babel ecosystem to dynamically coordinate the utilization of different sensors and actuators through a familiar (and unified) API, thus allowing them to easily integrate these devices into IoT and smart-home applications and other scenarios where distributed protocol execution and real-world interaction are required or otherwise useful, i.e. real-time monitoring.
These protocols and the underlying library make it possible to develop responsive yet highly complex and interactive distributed applications for a host of different scenarios and use cases.

Usage

Users can register events and notifications associated with specific sensor readings or actuator states. These events can then be propagated throughout the swarm of devices in active use, and from there define specific behaviors to be triggered upon their reception, enabling distribution coordination. This approach allows for real-time interaction and modular composition of behaviors, facilitating scalability and automation.

Additionally, collected information can be processed locally within the application by resorting to user-defined protocols. Processed results can then be propagated using the myriad available communication protocols already provided by the existing Babel framework.

Protocol API

The API provided for these interactions involves several different Babel-compatible protocols — two to control I^2C input and output devices, and two others to control Digital input and output, all of which rely on the Grove physical interface:

- **I²C:** Due to the nature of I²C devices, the protocols to control them follow a driver model, where unique implementations for each supported device and their behaviors are encapsulated within their interfaces. At the time of writing, the following devices are supported:
  - 3-Axis Accelerometer
  - Gesture Detector
  - 16x2 LCD
  - 8x8 RGB LED Matrix

- **Digital:** These protocols provide both a generic way to send and receive digital HIGH/LOW signals, as well as specific implementations for interacting with more complex devices such as Buzzers, Ultrasonic Rangers and Chainable RGB LEDs. The generic digital devices that have already been tested are as follows:
  - Line Finder
  - Tilt Switch
  - PIR Motion Sensor
  - Button

Designed with extensibility in mind, these protocols allow support for new devices to be added and developed without breaking changes or compatibility with existing implementations.
Moreover, the separation of input and output in the control protocols has a few distinct advantages:

- **Asymmetric resource allocation:** Users can independently assign behavior to each node without unnecessary overhead, based on a given node's primary function.
- **Scalable composition:** Complex behaviors can be designed by combining simple input-output relationships between nodes, allowing for more sophisticated system responses to emerge from relatively straightforward configurations.
- **Enhanced fault tolerance:** Partial sensor and actuator functions can remain operational even if components or entire nodes fail, by allowing redistribution of responsibility to alternative nodes.

TaRDIS Messaging Application — Preliminary Version Integration

A preliminary version of these protocols has already been integrated into the TaRDIS Messaging Application, where events can be triggered via the app to show text and display certain patterns in the aforementioned LCD and LED Matrix, as well as receive notifications triggered by the connected sensors.
A more comprehensive integration is planned but the current implementation serves as a proof of concept and helps validate the interoperability of these heterogeneous systems. Future iterations will expand support for additional device types and enrich the possibilities for interaction with existing ones.

3. A GUIDE TO THE TARDIS TOOLBOX WP6 COMPONENTS

Throughout the execution of the project, WP6 has produced several components to be integrated into the TaRDIS Toolbox, in addition to the Babel runtime variants (Babel-Swarm and Babel-Android) which provides the fundamental support for the execution of multiple of these components, taking advantage of the Java support across multiple devices (Servers, Laptops, Smaller devices such as Raspberry Pi and Intel Nuc, and even Mobile Android Devices). We should note that Babel was identified as a key innovation in the EU Innovation radar.

However, understanding how to take advantage of these different tools to build swarm applications can be a complex task. While WP3 is working on providing tools that can assist developers in picking some of these tools (an effort that is still ongoing), in this section we provide a brief guide for the main different available (at the time of writing) tools developed in WP6 identifying key aspects where they can be relevant. We plan to provide a more complete guide at the end of the project.

This presentation groups the tools around their base functionalities and aligns with the goals of the different tasks of WP6.

a. MEMBERSHIP ABSTRACTIONS

HyParView

HyParView is a decentralized, unstructured partial-view membership protocol designed to provide resilient and low-cost overlay maintenance. It operates using active and passive views to manage neighbor connections, enabling nodes to join, leave, or fail without compromising global connectivity. This reference implementation represents the base, unmodified version of the protocol, excluding any enhancements from Babel-Swarm such as discovery or autonomic control.

It is well-suited for dynamic environments or deployments using nodes with limited capabilities, and where a set of stable nodes can be configured statically as entry points (i.e., contact) for the swarm.

A reference implementation is provided as a Babel component.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/babel/protocols/membership/hyparview

HyParView with Discovery and Autonomic Management

This advanced variant of HyParView builds upon the base protocol with support for dynamic peer discovery and self-configuration. Nodes can discover peers automatically

using multicast or broadcast mechanisms and autonomously adjust internal parameters based on runtime metrics.

This makes it ideal for mobile or ad-hoc swarm systems where topology evolves rapidly and user intervention is infeasible.

A reference implementation is provided as a Babel component.

Repository: codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/protocols/membership/hyparview-with-discovery

HyParView with Security

This additional variant of HyParView builds upon the variant with discovery and autonomic management (described above) with support for fundamental security mechanisms, namely the use of encrypted communication channels and verifiable (through certificates) peer identity.

This makes it ideal for setting where notes have enough computational resources to handle the overhead of cryptography but where the probability of malicious nodes being introduced by an attacker is greater..

A reference implementation is provided as a Babel component.

Repository: codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/membership/secure-hyparview

X-BOT

X-BOT is an adaptive overlay protocol that reshapes the topology of an unstructured overlay to optimize specific performance metrics such as latency or bandwidth. It performs neighbor rewiring while maintaining randomization and fault tolerance.

X-BOT is particularly suited for scenarios where performance needs to be optimized dynamically, like interactive applications or geographically distributed swarms requiring efficient routing paths.

A reference implementation is provided as a Babel component.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/babel/protocols/membership/x-bot

X-BOT with Discovery and Autonomic Management

This advanced variant of X-BOT builds upon the base protocol with support for dynamic peer discovery and self-configuration. Nodes can discover peers automatically using multicast or broadcast mechanisms and autonomously adjust internal parameters based on runtime metrics.

This makes it ideal for mobile or ad-hoc swarm systems where topology evolves rapidly and user intervention is infeasible.

A reference implementation is provided as a Babel component.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/membership/x-bot-with-discovery

Epidemic Global Membership Protocol

This protocol builds a probabilistic approximation of global membership using epidemic dissemination on top of partial-view overlays. It aims to provide eventual convergence to a consistent global view under stable conditions.

It is particularly relevant for scenarios requiring system-wide visibility without strict consistency, such as swarm monitoring dashboards or adaptive control systems.

A reference implementation is provided as a Babel component.

Repository: codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/membership/epidemicglobalview

Random Tour Protocol

This protocol, while not strictly a membership protocol, enables decentralized estimation of network size using token-based random walks. Each token gathers statistics as it traverses the overlay, allowing individual nodes to infer global properties.

It is useful for parameter self-tuning in large-scale or elastic swarms, supporting protocols that need to adjust based on system size, such as adaptive gossip or routing schemes.

A reference implementation is provided as a Babel component.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/network-estimations/random-tour

b. Communication Primitives

Eager Gossip Broadcast

This gossip-based protocol supports probabilistic point-to-multipoint message dissemination. Upon receiving a message, each node forwards it to a subset of neighbors according to a configurable fanout, balancing redundancy with scalability.

It is well suited for scenarios with high churn or unstructured topologies, such as ad-hoc collaboration platforms, peer-to-peer gaming, or opportunistic mobile communication.

A reference implementation is provided as a Babel component.

Repository: codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/protocols/communication/eagergossipbroadcast

Self-Adaptive Eager Gossip Broadcast

This variant of the gossip-based protocol builds on the previously described version to add mechanisms that allow for the runtime-adaptation of the fanout employed by the protocol. This in turn allows the integration of this variant in settings where telemetry is collected, enabling the swarm to ensure that the fanout employed by nodes is adequate to the system size, and eventually the risk of disruptions in communication.

It is well suited for scenarios with high variability in the operational conditions of the application, or where communication mediums are less reliable.

A reference implementation is provided as a Babel component.

Repository: codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/communication/adaptive-eager-gossip-broadcast

Secure Eager Gossip Broadcast

This variant of the gossip-based protocol builds on the base protocol to add fundamental mechanisms for security, ensuring that communication channels are encrypted and messages are signed by their originators, enforcing authentication, non-repudiation, integrity, and privacy of communications..

It is well suited for scenarios where devices can handle the overhead of cryptographic primitives and where the data being conveyed by the application has some sensibility.

A reference implementation is provided as a Babel component.

Repository

codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/communication/eagergossipbroadcast-secure

:

Anti-Entropy Protocol

The anti-entropy protocol improves reliability by periodically synchronizing state among nodes to recover lost or missed messages. It is particularly valuable as a companion to probabilistic dissemination strategies like gossip.

This protocol is useful in environments with unreliable links, high message loss, or intermittent connectivity—such as mobile swarms or networks with constrained edge devices.

A reference implementation is provided as a Babel component.

Repository: codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/protocols/communication/antientropy

Flood Broadcast Protocol

The Flood Broadcast protocol is a simple, robust point-to-multipoint communication primitive that ensures reliable message dissemination across a swarm. It works by forwarding each broadcast message to all neighbors in the local view of each node, effectively flooding the overlay network. This approach guarantees that all connected nodes will eventually receive the message, assuming the underlying overlay remains connected and there are no persistent message losses.

Due to its simplicity and reliability, this protocol is well-suited for controlled environments with moderate node counts or low message volume, where reliability is more important than efficiency. It is particularly useful for bootstrapping operations, coordination in small swarms, or environments where message loss must be minimized, such as in mission-critical alerts or group coordination.

A reference implementation is provided as a Babel component.

Repository:
codelab.fct.unl.pt/di/research/tardis/wp6/babel/protocols/communication/floodbroadcast

One Hop Broadcast Protocol

A one-hop broadcast protocol is a communication protocol where a message is transmitted from a single sender to all of its directly connected neighbors (i.e., nodes that are one hop away) in a network.

For instance, a swarm sensor network, a node might use a one-hop broadcast to announce its presence or status to nearby nodes. These nodes receive the message directly, but they don't forward it further.

A reference implementation is provided as a Babel component.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/communication/onehopbroadcast

Anti-Entropy Protocol

The Anti-Entropy protocol is a reliable, periodic synchronization mechanism that allows nodes in a decentralized system to reconcile the messages that they have exchanged

through an epidemic gossip protocol. It operates by maintaining a local message history and periodically exchanging summaries or digests with neighbors to identify and request missing information. This ensures that, over time, all nodes in the network converge toward a consistent state, even if some messages were lost during initial dissemination.

This protocol is particularly useful in dynamic or unreliable environments where node failures, message drops, or temporary disconnections are common—such as mobile swarms, opportunistic networks, or systems operating over unstable wireless links. When combined with probabilistic dissemination protocols like gossip or broadcast, anti-entropy enhances reliability by guaranteeing eventual completeness of message delivery.

A reference implementation is provided as a Babel component.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/babel/protocols/communication/antientropy

codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/communication/antientropy

codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/protocols/communication/antientropy

Anti-Entropy Protocol with Security

The Anti-Entropy with Security protocol is built upon the previously described Anti-entrophy protocol and adds base authentication mechanisms for nodes (based on certificates) as well as encrypted communication channels, offering non-repudiation, integrity and privacy.

This protocol is particularly useful in setting where information exchanged at the application level has needs for the integrity and where malicious attacks can happen. Nodes should have enough computational power to handle the additional cryptographic overhead.

A reference implementation is provided as a Babel component.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/communication/anti-entropy-secure

Actyx Middleware Integration: Reliable and Durable Event Broadcast

This component integrates the Actyx middleware into the TaRDIS ecosystem to provide reliable and durable event broadcast capabilities. Originally a proprietary solution, Actyx was adapted and modularized to be usable as a reusable, open-source building block within swarm applications. The integration exposes a high-level event-stream abstraction that supports ordered, persistent, and fault-tolerant broadcast of application-level events, enabling developers to build swarm applications with stronger durability guarantees.

This solution is particularly relevant for scenarios requiring reliable audit trails, coordinated actions based on persisted events, or bridging disconnected segments of a swarm. It is suitable for use cases like industrial workflows, smart environments, or multi-agent systems where events must be retained and processed even after failures or network partitions.

c. Data Management Solutions

Arboreal

Arboreal is a hierarchical cloud-edge data management system that supports dynamic, fine-grained replication and causal+ consistency. It allows for seamless data migration and access optimization based on usage locality.

This solution is particularly relevant for smart infrastructure or industrial deployments that span across tiers, combining cloud analytics with low-latency edge processing.

Repository:

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

PotionDB

PotionDB is a geo-distributed, partially replicated storage system designed for eventual consistency. It supports materialized views and distributed querying, allowing applications to access local subsets of data with low latency.

This makes it ideal for mobile, edge, or hybrid swarm applications where bandwidth, storage, and energy efficiency are essential, such as smart city data aggregation or contextual services.

Repository:

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

Extensible CRDT Library for Babel

This library provides a collection of Conflict-free Replicated Data Types (CRDTs) tightly integrated into Babel. It supports a variety of data types and includes facilities for extending the library with custom CRDTs.

It is suited for collaborative applications that require concurrent updates without coordination, like decentralized logs, task managers, or shared data in offline-capable apps.

A reference implementation is provided as a Babel component.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-crdt/babel-crdts

Nimbus

Nimbus is a decentralized storage solution designed to support autonomous data sharing and ephemeral data retention in swarm environments. It enables nodes to locally manage the lifecycle of their data and selectively share information with peers based on configurable policies. The system avoids centralized control or full replication by relying on opportunistic and context-aware data dissemination mechanisms, making use of Babel's communication and membership abstractions for coordination. Nimbus was identified as a key innovation in the EU Innovation Radar.

Nimbus is particularly well suited for dynamic and resource-constrained environments where storage efficiency, privacy, and locality-awareness are critical—such as decentralized learning, collaborative sensing, and ephemeral swarms. It supports local-first semantics and is designed to operate in disconnected or intermittently connected networks, enabling resilient and adaptive data sharing strategies.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/internal-tools/nimbus

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

This component provides a set of adapters that expose the TaRDIS-defined storage API for integration with existing third-party or legacy distributed storage systems. It includes support for Cassandra, the blockchain-based IBM Hyperledger Fabric, C3 (a CRDT-based engine), and Engage (a previously developed system by consortium members). These adapters enable Babel-based applications to interact with these storage systems using a uniform interface, simplifying integration and testing.

This solution enables developers to combine new decentralized features with established data infrastructures, facilitating migration, hybrid deployments, or federated data management setups. It is useful in industrial, enterprise, or research environments that need to bridge swarm-native and existing storage backends.

A reference implementation is provided as a Babel component.

Repository:

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

Actyx Middleware Integration: Data Management for Event Streams

This component extends the Actyx middleware integration by enabling its use as a decentralized data management layer focused on event streams. It introduces support for event retention policies and stream-based querying, allowing swarm applications to process and analyze event histories efficiently. The integration transforms Actyx from a

standalone middleware into a modular, embeddable data service compatible with the TaRDIS vision of composable and distributed systems.

This tool is especially relevant for applications that rely on event sourcing or stream processing, such as reactive workflows, sensor networks, or multi-agent systems with asynchronous coordination. By supporting local event replay and on-demand query over persisted streams, it allows applications to make informed decisions even in the face of disconnections or limited infrastructure.

d. Monitoring Solutions

Namespace-Based Reconfiguration System

This system supports the decentralized and selective reconfiguration of application components using container-based isolation and hierarchical namespaces. It allows operators to apply reconfigurations based on logical structure and metadata labels.

It is applicable to complex, distributed deployments with legacy or black-box components where targeted updates, scaling, or parameter tuning must be automated.

Repository:

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

Docker Monitorization and Telemetry Acquisition

This tool extends the telemetry infrastructure by enabling fine-grained monitoring of swarm components running inside Docker containers. It leverages container introspection to collect runtime metrics such as CPU, memory, and network usage, and exposes telemetry for both system-level and container-scoped processes. Labels and metadata attached to containers are used to organize telemetry streams, allowing flexible and contextualized monitoring.

This solution is particularly valuable in containerized deployments where components may be ephemeral, heterogeneous, or orchestrated across dynamic environments. It supports automated resource management, anomaly detection, and deployment debugging in swarms composed of legacy and modernized components.

Telemetry Acquisition for Decentralised Systems

This component enables the decentralized collection of telemetry data across all layers of a swarm application—ranging from low-level protocol metrics and system resource usage to application-specific performance indicators. It is built as an extension of the Babel runtime and provides a unified interface for protocol and application developers to report telemetry data. Metrics are collected locally at each node, enabling both local analysis and export in standard formats for external processing.

This tool is particularly relevant for scenarios requiring runtime observability, performance tuning, or autonomic behavior. It supports debugging, real-time monitoring, and machine learning-driven adaptation, making it suitable for complex, long-lived, or mission-critical swarm systems. Its decentralized nature ensures that monitoring remains robust even under partial failures or in disconnected environments.

A reference implementation is provided as a Babel component.

Repository:

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

APPLICATIONS

In this section, we will present various applications that demonstrate how the tools introduced earlier can be effectively utilized. These examples will highlight the practical use and versatility of the methods discussed previously and their versatility to handle different use-cases. Although the functionalities of these applications are detailed in Demos, in this segment we focus on the tools that compose each.

Tardis Simple Usecase

The TaRDIS simple use case showcases a demonstrator of some of the core ideas governing the execution of the TaRDIS project. The application consists of a message exchanging application, where nodes communicate with each other in a completely decentralized way by dynamically discovering their neighbors.

The application uses HyParView with Discovery and Autonomic Management to handle membership and EagerPushGossip Broadcast, AntiEntropy and Random Tour to ensure reliable communication under the presence of failures.

Moreover, the application possesses two variants of the base implementation, namely a variant with a GUI for desktop environments, and a second one for mobile devices running Android.

Repository:
codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/applications/tardis-simple-usecase
codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/applications/tardis-simple-usecase-gui
codelab.fct.unl.pt/di/research/tardis/wp6/babel-android/applications/tardis-messaging-app

Decentralized Voting Application

The decentralized voting application allows users to evaluate presentations and workshops. Each user is able to rate different aspects of the lecture, as well as add new comments or questions to the lecture which can be later on rated by the other users that attend such presentations.

The application works in a completely decentralized fashion by using Nimbus as the underlying data storage solution. To achieve this, the implementation uses HyParView with Discovery and Autonomic Management to handle membership and node discovery, interfaced with Nimbus for the synchronisation of data.
The application can be interacted with by using a GUI for web-browsers or with a terminal through a REST API and Websockets for reactive notifications related to new events on the system (i..e., new questions, updates on ratings, etc).

Repository:
codelab.fct.unl.pt/di/research/tardis/wp6/internal-tools/nimbus/examples/sample-applications/-/tree/main/Presentation-App

OTHERS

In addition to the previously reported tools, we also have developed a library for IoT devices that has been integrated into the Babel ecosystem. This was a strategic move of our part, since IoT (and more generally domotics) applications are a perfect use case for the application of swarm technology, to improve the impact capacity of the project through specialised demonstrators in this domain, we have integrated a IoT device library for Raspberry Pi and Linux system compatible with the several variants of Babel.

Java IoT Library

The existing GrovePi library provides an open-source platform for connecting Grove sensors and actuators to Raspberry Pi devices. The Java portion of this library is outdated and extremely barebones, incapable of providing functional support for any particular device. The changes and significant additions made to it now allow for the integration of several I^2C and digital devices through a uniform interface.

Major additions to the library include the implementation of interfaces for interacting with the particular I^2C and digital devices highlighted in IoT Device Integration. While the main motivation for building this library was to support the Babel IoT Control Protocols, we note that our library is of interest for the community at large, providing a simple and effective way of interacting with a diverse set of IoT devices in the Java ecosystem.

Repository:

codelab.fct.unl.pt/di/research/tardis/wp6/iot/grovepi-pi4j

Babel IoT Control Protocols

As detailed in a previous section (IoT Device Integration), protocols were developed for providing control over IoT devices through a unified API within the context of the Babel-Swarm framework.

It can be used to support applications that interact with the physical world, namely those in the domains of IoT and Domotics, which are interesting application domains for the use of Intelligent Swarms, by allowing user devices to naturally interact with each other with minimal configuration, in a robust and efficient way..

A reference implementation is provided as a Babel component.

Repostory:

codelab.fct.unl.pt/di/research/tardis/wp6/iot/protocols/babel-iot-control-protocols

4. USE CASE PROGRESS REPORT

a. TELEFÓNICA

WP6 is collaborating with telefónica in the efforts to advance their existing federated machine learning solution (FLaaS) with decentralized interactions between nodes performing the training, and adding flexibility to exploit additional aggregators in a hierarchical solution and exploiting split learning techniques to be able to cope with (user) devices that have limited capabilities or energy, allowing these nodes to delegate part of the effort of training to helper nodes (either locally or centrally) without forcing those nodes to expose their input data.

Currently we have efforts ongoing to take advantage of the features in Babel-Swarm and Babel-Android specifically to assist in these efforts which combine research and engineering activities.

b. GMV

WP6 is collaborating with GMV to build a demonstrator of the application of TaRDIS solutions and technology to the use case of handling orbit determination in swarms of satellites. Evidently it is impossible (and unrealistic) in the time of the project to have the ideas and technology of TaRDIS effectively deployed on Satellites. Therefore our activities are fundamentally related to taking advantage of the high flexibility provided by the Babel ecosystem to create an emulation of these solutions in practice, where solutions will be running real software (not only within Babel) to build a simulacrum of what could happen in a real scenario between satellites.

Evidently, Babel was not designed to cope with the communication models employed for inter-satellite communication, and as such we are currently developing a specialized communication channel for Babel-Swarm that will try to approximate the conditions of these types of communication channels. Moreover, we are taking advantage of these specialized channels to build a model of the positions of satellites, that encodes specifically the restrictions of communication that might exist due to relative positions of these satellites.

Additionally we are building a specialized membership protocol that will interact with these special channels to expose information, in an opportunistic way, to other elements running within the device, this will include the Nimbus decentralised storage solution that will be leveraged to maintain navigational information collected among satellites and will replicate this information among them using CRDTs and an epidemic style protocol.

c. EDP

WP6 is involved in the preparation of the use case demonstrator of EDP by building a specialized membership protocol to organize the different participants of the decentralised renewable energy market in a way that is both location-aware (in the sense of proximity) and that allows an hierarchy (or several independent ones) to form among nodes in the same location. This will be used to model the notion of communities of energy that are fundamental local and where most energy exchanges (and hence coordination) happens. The hierarchy among these different communities can be leveraged in cases where the needs of elements of a community cannot be satisfied locally.

This development is being conducted in the context of Babel-Swarm, which already features several functionalities useful for the materialization of this use case. Furthermore, this offers the opportunity to collaborate with activities in the context of WP3, where DCR graph choreographies are being explored as a way to model the logic of the decentralized market of energy. There is an ongoing effort in WP3 to translate the logic denoted in DCR graphs to the Babel runtime.

d. ACTYX

WP6 has previously collaborated in the evolution of the use case of Actxy by exploring ideas to improve the Actyx framework to provide reliable and durable event broadcast as well as to come up with solutions for the durable storage of events propagated within the context of this framework. There are currently no other plans for WP6 to be involved in the use case of Actyx, which is justifiable since there is a proprietary framework at the center of this use case that has already imported some of the ideas and work of this work package.

Funded by the European Union

5. SOFTWARE

a. OVERVIEW

This section provides an overview of the main software components developed in the context of WP6 during the second year of the TaRDIS project. Rather than revisiting the numerous protocol implementations and abstractions that have already been discussed in detail in earlier sections of this deliverable—particularly in the context of Tasks T6.1, T6.2, and T6.3—this section focuses specifically on the major software artefacts and tools that have been designed, implemented, and made available as reusable components of the TaRDIS toolbox. These include frameworks and libraries that support the development, deployment, monitoring, and control of swarm applications, as well as supporting APIs and tools for integration with third-party systems and platforms. The goal of this section is to highlight the software engineering efforts underpinning these contributions and to provide references to their source code and usage documentation where applicable.

b. BABEL ECOSYSTEM

Link to relevant repositories:
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm
https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-android

Babel is a Java-based framework designed to simplify the development of distributed protocols by providing an event-driven programming model and robust abstractions for communication, concurrency, and protocol coordination. Initially developed outside the scope of TaRDIS, Babel supports modular and performant implementations of fault-tolerant distributed systems by abstracting low-level concerns such as message serialization, timers, and inter-protocol communication. Its API allows developers to define distributed protocols using state-machine-like semantics, and facilitates clean modularization through the use of channels, event queues, and thread-isolated protocol execution.

The TaRDIS project extends Babel through two significant evolutions: Babel-Swarm and Babel-Android. Babel-Swarm introduces runtime support for autonomic behavior, including dynamic adaptation and protocol reconfiguration. It integrates features such as self-monitoring, runtime feedback, and discovery of communication partners. This version allows developers to build decentralized protocols that are capable of adapting to environmental conditions and changing system objectives. Babel-Swarm can be built using standard Java tools and can be used to implement (standalone) applications that run on different devices running conventional operating systems. The typical way to build such an application is by taking advantage of the maven tool, importing the libraries for the Babel-Swarm-Core and all protocols required for the application operation, which we have decided to provide also as independent libraries that can be imported by maven. The code

for the Core of this variant of the framework and several protocols and examples are available at:
codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm

Babel-Android ports Babel's capabilities to the Android ecosystem, allowing mobile devices to participate in the execution of decentralized protocols, and swarm applications, natively. This extension adapts the Babel architecture to support mobile limitations, including supporting background execution. It enables the development of Android apps that incorporate swarm-native communication and logic, and has been used to prototype native android swarm applications that can interact with applications in commodity devices in TaRDIS. The programming model is similar to that of Babel-Swarm, although some specific examples of how to encapsulate specific protocols and the Babel-Android core in a foreground service are also provided.
The source code for Babel-Android, several protocols, and relevant applicational examples are available at:
codelab.fct.unl.pt/di/research/tardis/wp6/babel-android

We omit here the description and listing of all protocols and components developed in the context of the Babel ecosystem since these are already reported previously in A GUIDE TO THE TARDIS TOOLBOX WP6 COMPONENTS.

c. Reconfiguration and Monitorization tool based on Namespaces

Link to 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 already installed containerized tools *Docker* and *Docker Compose*. For *Windows* users, the one more prerequisite must be met, a *Unix*-like environment and command-line interface tool (e.g. git bash for Windows, *Cygwin*) must be already installed. All mentioned tools are open source, and free to download.

After 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. In the future we plan to simplify this process, and remove the cloning of the source code and just start containerized services that are pulled from *Docker Hub*, or other public container registry.

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 *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 *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. 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.

To access basic interaction of swarm and/or resources and basic metrics information users must use the CLI tool already present in the repository. This tool first must be prepared for a machine running on. To do that, users must first install the Golang environment on their specific machine which is free. Navigate to the *cockpit* project directory using command *cd cockpit*. After that user must build the tool using the command *go build -o cockpit*. To interact with CLI there is extensive documentation already present in the repository. If users do not want to use the CLI tool, and want to use some other option (e.g. Postman) there is extensive documentation in the repository for each service the tool offers.

Sample *Grafana* dashboards are available at the address https://localhost:3000 The default credentials are used for authentication (admin/admin). Dashboards are preconfigured, thus ready to use out of the box. Their configuration is available at this repository. The *Node Metrics* dashboard displays all available metrics for a selected node, while the *Container Metrics* dashboard contains charts for the most important metrics of selected or all containers running inside nodes.

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.

6. DEMOS

A. TaRDIS Messaging App

The TaRDIS Messaging App is a broadcast messaging application designed to showcase the capabilities of swarm-based communication systems built using the TaRDIS toolbox. Its goal is to provide a concrete and simple example of how autonomous, decentralized systems composed of heterogeneous nodes can support robust and adaptive message dissemination. The application allows participants to send and receive messages — optionally with file attachments — that are disseminated to all nodes in the swarm. Demonstrations of the app are conducted across a variety of platforms, including headless Raspberry Pi devices, desktop computers with a graphical user interface, and Android smartphones.

At the core of the Messaging App lies a decentralized overlay network built using a self-configuring version of the HyParView protocol. This overlay provides the basis for communication among the swarm participants and is constructed automatically upon a node's entry into the system using multicast-based discovery. Once connected, nodes begin participating in a fully distributed gossip-based broadcast protocol that ensures all messages are eventually received across the network. To improve reliability and support resilience to transient faults and message loss, a periodic anti-entropy protocol is used to reconcile the messages received among neighbors. Additionally, the application includes an implementation of the random tour protocol that allows each individual node to estimate the current size of the swarm.

The Figures below show a simple representation of the combination of protocols and components within Babel-Swarm or Babel-Android for the different variants of the application. Notice that there is a linux-headless version that runs on raspberries, and a version for laptops and android devices with interface. Some devices use a special version of the application that can interact with IoT devices. These heterogenous applications can however cooperate and collaborate in the context of this swarm.

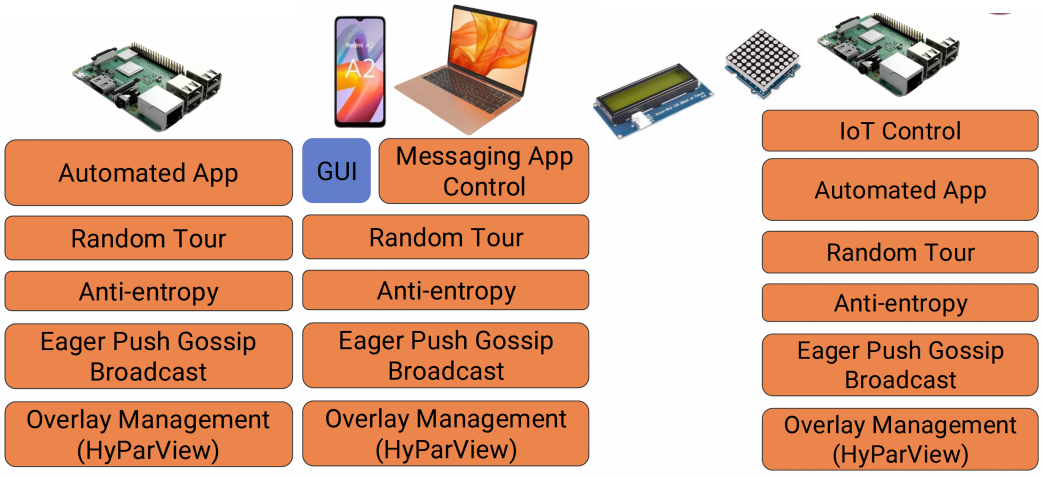


Fig: Internal composition of Tools for materializing different version of the App.

The application was implemented entirely in Java and relies on the Babel-Swarm and Babel-Android frameworks developed in WP6. These frameworks extend the Babel programming model with self-management and self-configuration capabilities, enabling seamless deployment and operation across devices with very different capabilities, from lightweight embedded boards to fully featured Android platforms.

The Raspberry Pi instances of the application run as a background Linux service and include an autonomous mode that periodically generates and broadcasts messages. In contrast, the desktop version provides a graphical user interface that allows users to compose messages, attach files, and interactively follow the status of dissemination and swarm connectivity. Android support further extends the reach of the app, with a mobile-friendly interface and integration with the local wireless interface for ad hoc participation in swarms. The figures below show examples of the graphical interfaces for both laptops and android devices.

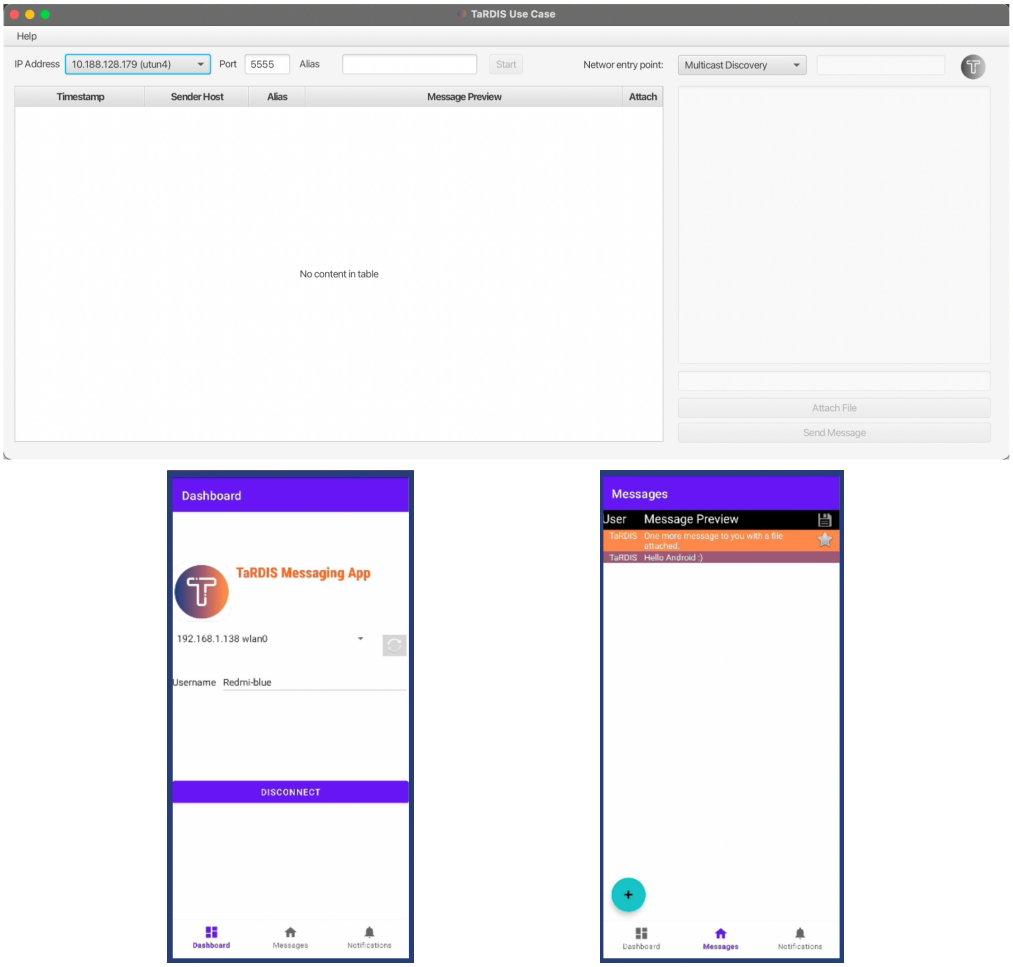


Fig: Interfaces provided for different versions of the TaRDIS Messaging App

The demo has been shown operating in a confined wireless local network environment (TaRDIS-LAB), although it is designed to function at global scale given appropriate network configurations (e.g., public IPs or NAT forwarding). In addition to demonstrating message exchange, the application has also been used to showcase the control of IoT devices

integrated with the swarm. In its latest version, specific keywords in messages can trigger effects on connected displays or LEDs, using a Babel-based control protocol to manage physical devices (discussed in Other Integrations and Activities).

For demonstration purposes, the application can be run on any machine with Java 21 or later by downloading the JAR file available at:
https://asc.di.fct.unl.pt/~jleitao/tardis.jar

Android versions are available for devices running Android 12 or higher via:
https://novasys.di.fct.unl.pt/packages/apks/

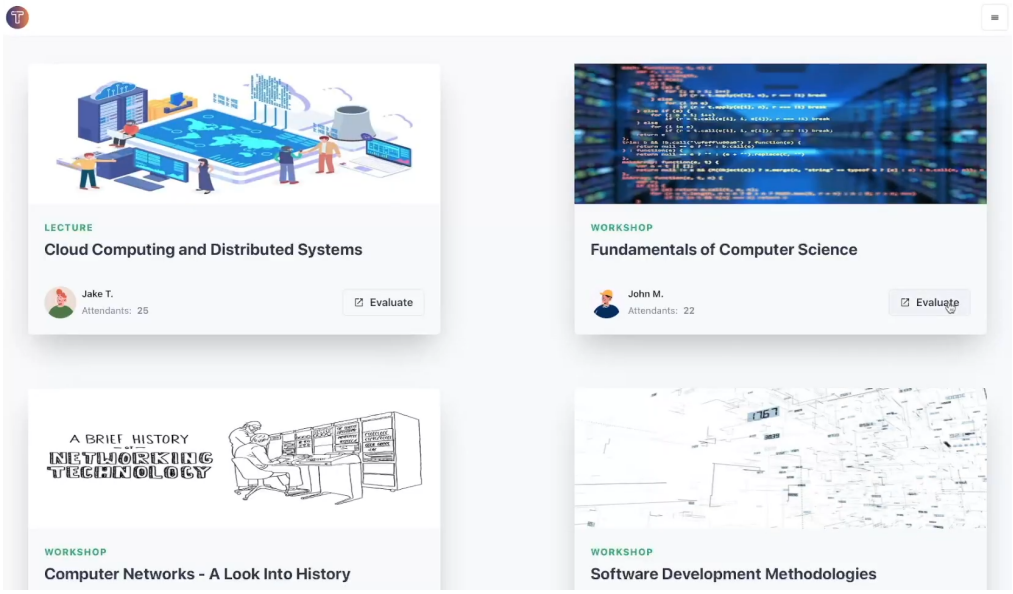
This demo exemplifies the potential of TaRDIS to support dynamic, fault-tolerant, and infrastructure-independent applications that require autonomous behavior and real-time coordination among decentralized nodes.

B. TaRDIS Voting App

In order to showcase the functionalities provided by Nimbus, we developed a demonstration application that enables users to evaluate presentations/workshops. The different presentations are constituted by different questions that can be rated and voted by users. Users can add new questions and see the ratings, through an average, of the different questions (i.e., was the information useful?) attached to the presentation.

This application works by leveraging Nimbus decentralized storage system, and thus having each user store the data of the application locally, without relying on any dedicated infrastructure. Each node/participant in the system stores the information on their device, and the information is synchronized in the background.

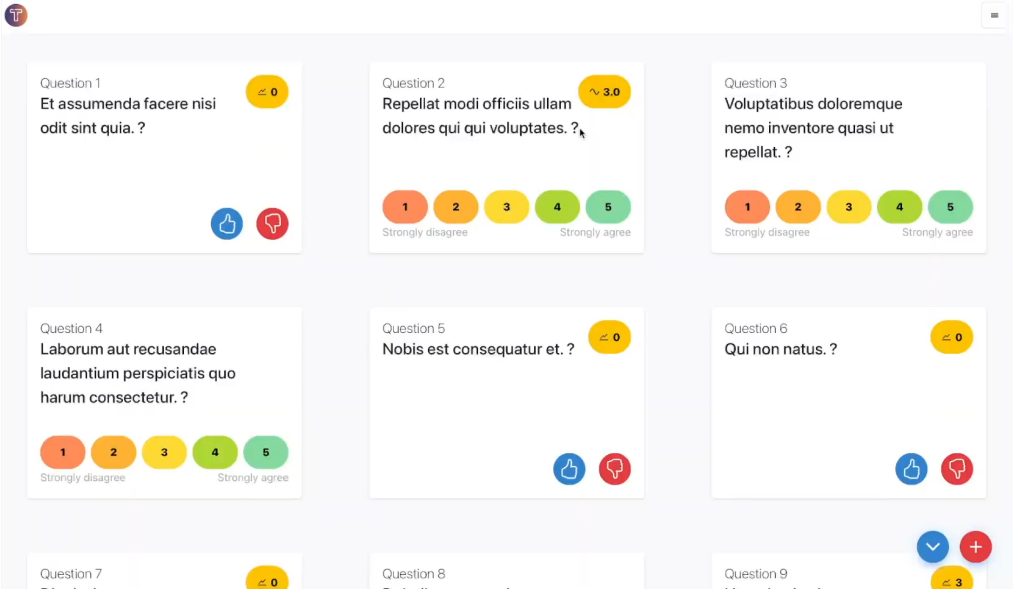
The demonstration is composed of a UI for web browsers and a deployment of Nimbus per node in charge for storing and synchronizing the application data. To allow the flow of data between the user interface (i.e., web browser) and Nimbus (i.e., in the Babel ecosystem) we leverage the Babel API for Web Services, to forward the requests issued by the user to Babel. Moreover, when new information arrives to a node, the UI is updated accordingly in real time by using the previously established Web Socket (more details can be found in Babel API for Web Services).



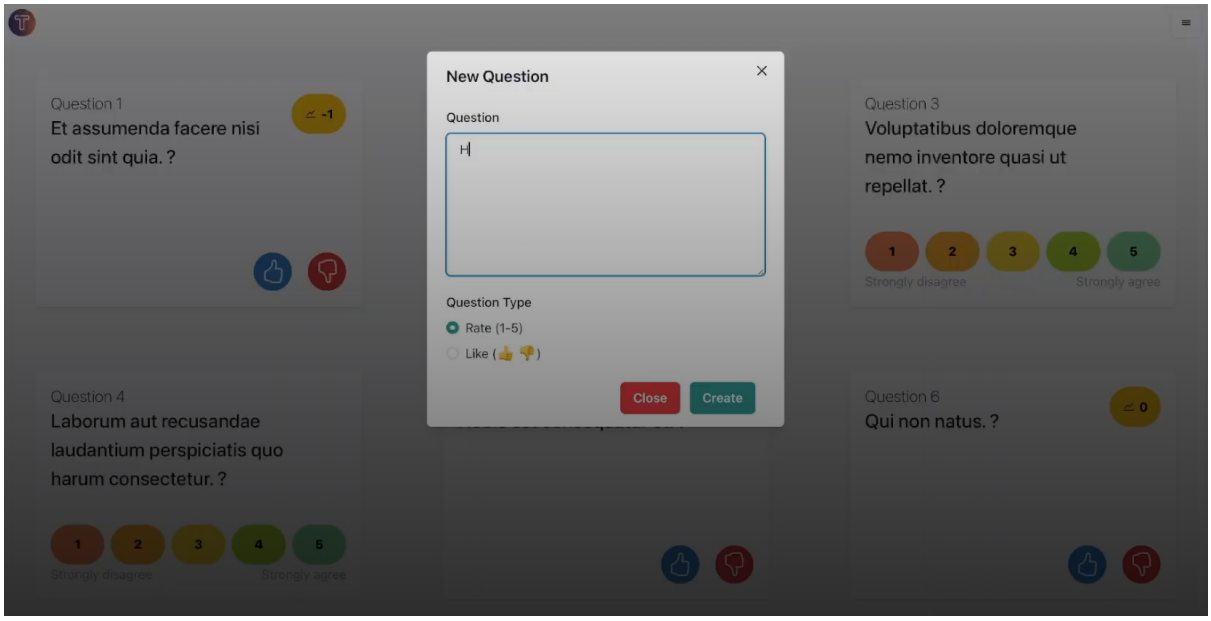
TaRDIS Voting App - Home page

To simulate a real word swarm system, the application was deployed in heterogeneous machines, namely Raspberry Pi's and laptops, by allowing nodes to enter or leave the swarm system at will. Users can interact with the different devices by entering the local network where the application is deployed, and accessing the web site being serviced by the nodes through their favorite device (i.e., smartphone, tablet, etc.), or simply by joining the swarm system with their own device.

Upon entering the website, users can login with their unique username and see the plethora of presentations available for voting. Upon entering a presentation, the user is presented with a list of available questions, the option to vote on them and to add new questions that will be presented to all users participating in that presentation.



TaRDIS Voting App - Voting page



TaRDIS Voting App - Add question page

This results in highly dynamic and decentralized systems, where no device controls all the flow of information and where each user can interact with the system in a seamless way without having to concern with the complexity being used underneath.

Status

At the moment of writing, this demonstration is being updated to support the new functionalities of Nimbus, namely, partial replication, reconfiguration of keyspaces and collections and access control.

Repository

An working example of this demonstration can be seen in the demonstration video[17], and a the full code snippets can be found on the Nimbus - Examples[18] and Presentation Demo UI[19].

[17] https://drive.google.com/file/d/1PWkXfJEeeYiroZvePUjFFgSVuSWI_Wm0/view?usp=sharing
[18]https://codelab.fct.unl.pt/di/research/tardis/wp6/internal-tools/nimbus/examples/sample-applications/-/tree/76b9e3c04716585e1e18964b9c351c288996cb56/Presentation-App
[19] https://codelab.fct.unl.pt/di/research/tardis/wp6/demonstrations/presentation-evaluation-demonstration/-/tree/7ca32d11f73fba24fd393b22829f4debffd9c4d3

7. STATE OF THE ART REVISION

a. Frameworks for Developing Decentralized and Swarm Systems

The design and deployment of decentralized swarm systems present unique challenges that go beyond traditional distributed systems, including self-organization, high churn resilience, and adaptability to resource-constrained environments. Several frameworks have been proposed to support these goals. Appia [24] offers composable protocol stacks using Java, but suffers from rigidity due to strict protocol stacking and a single-threaded execution model. Cactus [25], developed in C++, provides high performance through asynchronous meta-protocols but leaves concurrency management to the developer. Yggdrasil [26], a lightweight C-based framework, was specifically designed for handling ad hoc and wired networks and supports concurrent protocol execution, but is limited to a single network interface and demands low-level programming discipline. ViSiDia [27] serves mainly for educational purposes by simulating distributed algorithms but lacks scalability or production-grade support. The Babel framework stands out for combining modularity, an event-driven model, and extensible networking, facilitating the development of interoperable and reusable distributed protocols

.

Recent evolutions conducted in the context of TaRDIS such as Babel-Swarm further enhance this approach by integrating security mechanisms, dynamic peer discovery, autonomic management, and the Babel-Android variant introduced native Android compatibility, a set of unique properties in this field that can benefit modern and emerging swarm systems operating in open, dynamic, and heterogeneous environments.

b. Decentralised Membership and Communication Primitives

Membership and communication are foundational services in decentralized systems. The literature distinguishes between peer sampling services, which provide random node views suitable for gossip dissemination, and overlay networks, which create structured topologies to support scalable routing and message exchange.

Well-known protocols such as HyParView [17] provide resilient overlays using active and passive views to ensure robust connectivity and churn resilience. Additional decentralized membership abstractions exploit other approaches, such as Cyclon [29] that promotes a more dynamic membership abstraction, or Scamp [30] that strives to balance the load automatically between nodes in a more localised fashion. Communication abstractions such as epidemic dissemination [17], anti-entropy, and spanning tree-based dissemination [28] offer trade-offs between redundancy, fault tolerance, and performance. However, many of these primitives are tightly coupled to specific implementations or APIs, limiting their reusability or interchangeability in application development, or putting a significant burden on developers to implement them in their own applications.

TaRDIS has unified multiple classes of membership and communication protocols — such as HyParView, X-BOT [20], eager gossip, anti-entropy, and flood broadcast — within a composable API architecture provided in the context of the Babel framework and its evolution (also achieved in the context of TaRDIS). It has also introduced novel abstractions, including the epidemic global membership service and evolved existing solutions (namely HyParView) with new functionalities that simplify their usage across different settings.

c. Distributed Data Management System

We divide the related work overview in three parts, focusing on each of the works reported in this deliverable.

Distributed queries.
A large number of databases for cloud settings has been designed in recent years. Some of these databases are mostly designed for supporting full-replication, where each site replicates the full database, such as Spanner [7]. In other systems, each site might not replicate the full database, making them closer to our work.

Detok [8], L-Store [9] and DynaMast [10] support transaction execution over geo-partitioned data by using data migration and dynamic remastering to guarantee single-partition transactions. This is similar to our approach to support multi-site transactions. However, these systems have no support for efficiently handling recurrent queries.

CouchDB [11] provides both indexes and materialized views, however views can only refer to data in one partition. ChronoCache [12] is a caching middleware for geo-replicated databases. While caching speeds up future requests, complex queries may still need to contact multiple servers and updates need to invalidate the cache or else clients read stale data. We have proposed an approach providing consistent and efficient query results.

Decentralized data stores
PeerDB [16] is a peer-to-peer distributed data sharing system. It was designed to be used in a web setting, in which every node can communicate with each other. This might not be the case in swarm applications, in which communication may be restricted. We have proposed an approach where nodes find and connect to each other in a dynamic way.

OrbitDB [15] is a serverless, distributed, peer-to-peer database. OrbitDB uses web3 IPFS as its data storage and automatically syncs database replicas. It is an eventually consistent database that uses a variant of CRDTs, Merkle-CRDTs, for merging concurrent updates. Due to this, all data on OrbitDB is immutable, thus guaranteeing a transparent and verifiable ledger. Nimbus shares some of the goals and design choices with OrbitDB, but it was designed to support the dynamicity and scale of swarm applications. Contrary to OrbitDB which uses a DHT (Distributed Hash Table) to discover peers. Nimbus uses a more dynamic approach (i.e., multicast, mDNS, etc.) for finding peers. Moreover, data in Nimbus is mutable, therefore enabling the use of dynamic applications.

CRDT library

A number of CRDT libraries exist. YJS [13] (https://yjs.dev) is a shared editing framework for the JavaScript ecosystem. It features a CRDT library that can be used to build collaborative editing applications and binding to different frameworks and networking libraries. Our Babel library is specific for the Babel framework and designed to address the specific needs of swarm applications. Developers may choose from a list of already available CRDTs (i.e., sets, maps, flags, etc.) or develop their own handmade CRDTs by extending the abstractions provided by the library.

Automerge [14] is a library of data structures for building collaborative applications with a JavaScript and a Rust implementation. Similarly to YJS, it is also network library agnostic, but it is designed for being used in a Web setting. On the contrary, our Babel library is designed to be used by swarm applications, in an environment that is expected to be much more dynamic.

d. Decentralised Monitoring

Decentralized monitoring is crucial for observing and analyzing distributed systems without relying on centralized entities, thereby enhancing scalability and fault tolerance. Traditional centralized monitoring approaches can create bottlenecks and single points of failure. To address these challenges, decentralized monitoring algorithms have been developed. For instance, Kim et al. introduced an efficient decentralized monitoring algorithm that checks for violations of safety properties in distributed programs [31]. Previous work has proposed methods to monitor decentralized specifications, distinguishing between centralized and decentralized approaches [32]. Additionally, Olston et al. explored adaptive filters for continuous queries over distributed data streams, contributing to decentralized real-time monitoring techniques [33]. Closer to what we are doing in TaRDIS is the work reported in [34], that is a small exploratory work that leverages overlay networks to monitor the activity of nodes in a decentralized fashion.

The TaRDIS project advances decentralized monitoring by aiming at developing fine-grained telemetry acquisition directly into the Babel framework having the potential to combine that data with indicators from the local device and operating system. This integration enables real-time, decentralized collection of system and application-level metrics, facilitating adaptive monitoring strategies that are resilient to node failures and network partitions. By embedding monitoring capabilities within the communication substrate, TaRDIS ensures robust and efficient monitoring, even in highly dynamic swarm environments.

e. Self-Management of Distributed Systems

Self-management in distributed systems enables autonomous configuration, optimization, healing, and protection without human intervention. The complexity of modern distributed computing necessitates such autonomic behaviors to maintain optimal performance. Kephart and Chess articulated the vision of autonomic computing, emphasizing the need for systems capable of self-management to address growing complexities [35]. Kramer and Magee discussed architectural challenges in designing self-managed systems, highlighting the importance of software architectures that support self-adaptive behaviors [36]. Additionally, Ghosh et al. proposed a control-based framework for self-managing distributed computing systems, focusing on continual performance optimization [37]. Despite these efforts, and the long time since the initial proposal by IBM for autonomic systems, this vision has not yet been materialized in the context of distributed systems, and particularly decentralized systems, in an impactful way.

TaRDIS enhances self-management by providing a modular and extensible framework that supports dynamic reconfiguration and adaptation of swarm applications. By integrating autonomic management features within the Babel framework, TaRDIS is striving to address a significant challenge, of practical self-management of complex systems. Additionally, TaRDIS is striving to bring machine learning models to guide reconfiguration of systems that, due to their complexity, are inherently hard to reason about, even by domain experts.

f. Decentralized Machine Learning

Decentralized Machine Learning (DML) facilitates training models across multiple nodes without central coordination, preserving data locality and enhancing privacy. Federated Learning (FL) is a prominent paradigm in this domain, enabling collaborative AI training across organizations without compromising data privacy. DFLStar, a decentralized federated learning framework incorporating self-knowledge distillation to enhance local model training and optimize communication overhead [38] has been proposed recently trying to improve the quality of decentralized federated learning approaches. Additionally, Belal et al. introduced PEPPER, a decentralized recommender system based on gossip learning principles, demonstrating improved convergence speed and recommendation performance [39].

TaRDIS contributes to DML by integrating decentralized machine learning capabilities within its ecosystem, leveraging the Babel framework's communication primitives to facilitate efficient model dissemination and aggregation. This integration aims at supporting collaborative learning in swarm environments, enabling nodes to train models locally and share updates in a peer-to-peer fashion. By embedding DML within a robust decentralized framework, TaRDIS enhances the scalability, privacy, and adaptability of machine learning applications in dynamic and heterogeneous environments, making it more available and easier to use, even in settings where user data used in the training lies on mobile devices.

8. PUBLICATION AND DISSEMINATION ACTIVITIES

a. 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:

[1] M. Simić, J. Dedeić, M. Stojkov and I. Prokić, "Data Overlay Mesh in Distributed Clouds Allowing Collaborative Applications," in IEEE Access, vol. 13, pp. 6180-6203, 2025, doi: 10.1109/ACCESS.2024.3525336.
[2] T. Ranković, F. Šiljić, J. Tomić, G. Sladić and M. Simić, "Misconfiguration Prevention and Error Cause Detection for Distributed-Cloud Applications," 2024 IEEE 22nd Jubilee International Symposium on Intelligent Systems and Informatics (SISY), Pula, Croatia, 2024, pp. 000297-000302, doi: 10.1109/SISY62279.2024.10737513.
[3] M. Simić, J. Dedeić, M. Stojkov and I. Prokić, "A Hierarchical Namespace Approach for Multi-Tenancy in Distributed Clouds," in IEEE Access, vol. 12, pp. 32597-32617, 2024, doi: 10.1109/ACCESS.2024.3369031.
[4] Ranković, T., Kovačević, I., Maksimović, V., Sladić, G., Simić, M. (2024). Configuration Management in the Distributed Cloud. In: Trajanović, M., Filipović, N., Zdravković, M. (eds) Disruptive Information Technologies for a Smart Society. ICIST 2024. Lecture Notes in Networks and Systems, vol 860. Springer, Cham. https://doi.org/10.1007/978-3-031-71419-1_20.
[5] Kovačević, I., Ranković, T., Simić, M., Stojkov, M. Token-based identity management in the distributed cloud. In: Zdravković, M., Trajanović, M., Filipović, N., Konjović, Z. (Eds.) ICIST 2024 Proceedings, pp.152-161, 2024.
[6] P. Fouto, N. Preguiça and J. Leitão, "Large-Scale Causal Data Replication for Stateful Edge Applications," 2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS), Jersey City, NJ, USA, 2024, pp. 209-220, doi: 10.1109/ICDCS60910.2024.00028.
[7] Nuno Policarpo, José Fragoso Santos, Alcino Cunha, João Leitão, Pedro Ákos Costa. Specifying Distributed Hash Tables with Allen Temporal Logic Proceedings of FormaliSE 2025 (co-located with ICSE), Ottawa, Ontario, Canada, 2025 (to appear).

b. DISSEMINATION ACTIVITIES

- Participation in the “Swarms Projects Workshop” in Bruxelles, September 2024. João Leitão was there representing TaRDIS as discussing the technologies being put forward by TaRDIS in general and WP6 in particular.

9. 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, WP7 and WP8).

a. WP3

WP3 is currently collaborating with WP5 to achieve two relevant results. The first is the translation of DCR-Graph choreographies to the Babel runtime, materializing the logic defined in this format to a set of distributed protocols and their orchestration. Additionally, exploratory work is being carried out to build a tool that can provide suggestions of the most adequate protocols to be used (from the ones available in the Babel ecosystem) given the needs of a specific application.

Additionally, the IDE being developed in the context of WP3 has already had support added to assist developers of protocols and applications using the Babel framework.

b. WP4

WP6 developed a centralised reconfiguration engine based on hierarchical namespaces to manage components across diverse devices using containerisation. When developing such a tool, WP6 needed strong guarantees that redistribution of resources is formally correct, while the developed protocols ensure correctness. The developed engine relies on: (i) accurate resource redistribution, which is achieved by applying graph transformation theory, and (ii) communication protocols, which are modeled and validated for correctness using multiparty session types. Both elements were developed in active collaborations with WP4.

c. WP5

For the particular case of WP5, WP6 has several contact points and active collaborations that have resulted in relevant results for the TaRDIS project.
Underlying the efforts to improve the decentralized machine learning, and in close alignment with the use case of Telefónica, we are exploring mechanisms using the mechanisms provided by Babel-Swarm to improve decentralized machine learning frameworks, enriching them with split-learning, fault-tolerance, and self-configuration.
Considerable efforts were made by WP6 to provide data sources for machine learning model training being carried out in the context of WP5.
Keeping in line with current efforts to integrate results generated by TaRDIS, PTB-FLA was adapted to take advantage of the communication primitives provided in the context of the Babel ecosystem.

d. WP7

There are several activities being carried out in the context of WP6 that aim to provide additional tools for the development of the use case demonstrators that are being carried out in the context of (more precisely coordinated by) WP7. The most relevant examples have been discussed in this document and include the membership abstraction exploiting node locality and hierarchies for the EDP use case, as well as the membership abstraction that can deal with the dynamic communication model of satellites for the GMV use case. Additionally, WP6 is highly involved in the preparation of the demonstrator for the latter use case, that will rely on emulation, taking advantage of other tools built in the context of WP6 such as the Nimbus decentralized data management system.

e. WP8

WP8 is responsible for dissemination, exploitation, and standardization. As part of this effort, WP6 has been actively engaged in standardization activities.

Currently, WP6 is drafting an Internet-Draft that was recently submitted to the Internet Engineering Task Force (IETF). This draft[20] outlines the architecture of a generic framework for building decentralized dynamic systems and aims to expand and open the Babel framework to new developers interested in swarm system development. This effort aims to pave the way for the standardization of the tools being developed in WP6 and its adaptation by external entities.

At the time of writing, the draft is under internal review and is set to be presented and submitted at the upcoming IETF 123 meeting in Madrid. As part of this initiative, TaRDIS will participate in the event's hackathon, introducing the draft and its reference implementation to newcomers and encouraging contributions to its development.

WP6 will continue to work on this document to refine its specifications, as well as writing new internet drafts to explain certain aspects of the framework in further detail.

[20] https://datatracker.ietf.org/doc/draft-jesus-gfds/

10. CONCLUSIONS

This deliverable presents the progress achieved in Work Package 6 (WP6) of the TaRDIS project during its second year. WP6 plays a central role in the development of the TaRDIS toolbox by designing and implementing the core runtime support for decentralised swarm systems. During this reporting period, the work focused on extending and consolidating the foundational abstractions introduced in the first year, with substantial advances across all three technical tasks.

New and enhanced protocol implementations were produced in the context of decentralised membership, communication, and monitoring. These were integrated into the Babel ecosystem and validated through cross-platform support, including mobile and embedded devices. The autonomic runtime introduced by Babel-Swarm, and its adaptation for Android through Babel-Android, enabled the deployment of adaptive, UI-less applications across heterogeneous swarms. At the same time, the data management stack was evolved through the consolidation of PotionDB and Arboreal, and the creation of a novel fully decentralised solution, based on CRDTS, named Nimbus. In the area of monitoring and reconfiguration, new tools for telemetry acquisition, metric aggregation, and decentralised reconfiguration were introduced and validated through demonstrators.

The outcomes of WP6 during this period are not limited to research contributions but also include mature software components that are publicly available and ready to be reused, integrated, or extended. These components were validated through demonstrators that showcase the runtime's ability to operate across heterogeneous environments, including mobile, embedded, and cloud-based nodes.

In the final year of the project, WP6 will continue its efforts by finalising the unified runtime implementation of Babel 2, expanding the toolbox with new mechanisms for secure communication and decentralised intelligence, and preparing the runtime for integration with the full set of TaRDIS use cases. The goal is to deliver a coherent, robust, and reusable runtime environment for building the next generation of intelligent and adaptive decentralised systems.

Finally, we note that two of the outcomes of WP6, namely the Babel ecosystem and Nimbus, have been identified as key innovations by the EU Innovation Radar.

Funded by the European Union

REFERENCES

1. Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In Proceedings of the 26th International Conference on World Wide Web (WWW '17). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 283–292. https://doi.org/10.1145/3038912.3052673
2. P. Fouto, P. Á. Costa, N. Preguiça and J. Leitão, "Babel: A Framework for Developing Performant and Dependable Distributed Protocols," 2022 41st International Symposium on Reliable Distributed Systems (SRDS), Vienna, Austria, 2022, pp. 146-155, doi: 10.1109/SRDS55811.2022.00022.
3. Gonçalo Cabrita and Nuno Preguiça. 2017. Non-uniform Replication. In Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS 2017).
4. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. A comprehensive study of convergent and commutative replicated data types. (2011).
5. Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao 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.
6. 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. 385–400.
7. James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. ACM Trans. Comput. Syst. 31, 3, Article 8 (August 2013), 22 pages. https://doi.org/10.1145/2491245
8. Cuong D. T. Nguyen, Johann K. Miller, and Daniel J. Abadi. 2023. Detock: High Performance Multi-region Transactions at Scale. Proc. ACM Manag. Data 1, 2, Article 148 (June 2023), 27 pages. https://doi.org/10.1145/3589293
9. Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 1659–1674. https://doi.org/10.1145/2882903.2882923
10. Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive Dynamic Mastering for Replicated Systems. In 2020 IEEE 36th International Conference on Data Engineering (ICDE). 1381–1392. https://doi.org/10.1109/ ICDE48307.2020.00123
11. Apache Software Foundation. 2024. ApacheCouchDB® 3.4.2 Documentation. https://docs.couchdb.org/en/stable/.

12. Brad Glasbergen, Kyle Langendoen, Michael Abebe, and Khuzaima Daudjee. 2020. ChronoCache: Predictive and Adaptive Mid-Tier Query Result Caching. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 2391–2406.
13. Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2015. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In Proceedings of the 15th International Conference on Engineering the Web in the Big Data Era - Volume 9114 (ICWE 2015). Springer-Verlag, Berlin, Heidelberg, 675–678. https://doi.org/10.1007/978-3-319-19890-3_55
14. OrbitDB. **https://dbdb.io/db/orbitdb**. 2025
15. W. S. Ng, B. C. Ooi, K. . -L. Tan and Aoying Zhou, "PeerDB: a P2P-based system for distributed data sharing," Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405), Bangalore, India, 2003, pp. 633-644, doi: 10.1109/ICDE.2003.1260827. keywords: {Data engineering},
16. Joao Leitão, Pedro Ákos Costa, Maria Cecília Gomes, and Nuno Preguiça. Towards Enabling Novel Edge-Enabled Applications. Technical Report. May 2018. https://asc.di.fct.unl.pt/~jleitao/pdf/NewEdgeApplications.pdf
17. J. Leitao, J. Pereira and L. Rodrigues, "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast," 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), Edinburgh, UK, 2007, pp. 419-429, doi: 10.1109/DSN.2007.56.
18. Laurent Massoulié, Erwan Le Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. 2006. Peer counting and sampling in overlay networks: random walk methods. In Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing (PODC '06). Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/1146381.1146402
19. João Leitão. Topology Management for Unstructured Overlay Networks. Ph.D.'s Thesis, Technical University of Lisbon, 2012.
20. J. Leitão, J. P. Marques, J. Pereira and L. Rodrigues, "X-BOT: A Protocol for Resilient Optimization of Unstructured Overlay Networks," in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 11, pp. 2175-2188, Nov. 2012, doi: 10.1109/TPDS.2012.29.
21. J. C. A. Leitao, J. P. d. S. F. M. Marques, J. O. R. N. Pereira and L. E. T. Rodrigues, "X-BOT: A Protocol for Resilient Optimization of Unstructured Overlays," 2009 28th IEEE International Symposium on Reliable Distributed Systems, Niagara Falls, NY, USA, 2009, pp. 236-245, doi: 10.1109/SRDS.2009.20.
22. J. C. A. Leitão and L. E. T. Rodrigues, "Overnesia: A Resilient Overlay Network for Virtual Super-Peers," 2014 IEEE 33rd International Symposium on Reliable Distributed Systems, Nara, Japan, 2014, pp. 281-290, doi: 10.1109/SRDS.2014.40.
23. M. Haridasan and R. van Renesse, "Defense against Intrusion in a Live Streaming Multicast System," Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06), Cambridge, UK, 2006, pp. 185-192, doi: 10.1109/P2P.2006.15.
24. H. Miranda, A. Pinto and L. Rodrigues, "Appia, a flexible protocol kernel supporting multiple coordinated channels," Proceedings 21st International Conference on

Distributed Computing Systems, Mesa, AZ, USA, 2001, pp. 707-710, doi: 10.1109/ICDSC.2001.919005.

25. N. T. Bhatti. A system for constructing configurable highlevel protocols. PhD thesis, University of Arizona, 1996.
26. Pedro Ákos Costa, André Rosa, and João Leitão. 2020. Enabling wireless ad hoc edge systems with Yggdrasil. In Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC '20). Association for Computing Machinery, New York, NY, USA, 2129–2136. https://doi.org/10.1145/3341105.3373908
27. Wahabou Abdou, Nesrine Ouled Abdallah, and Mohamed Mosbah. 2014. ViSiDiA: A Java Framework for Designing, Simulating, and Visualizing Distributed Algorithms. In Proceedings of the 2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications (DS-RT '14). IEEE Computer Society, USA, 43–46. https://doi.org/10.1109/DS-RT.2014.14
28. J. Leitao, J. Pereira and L. Rodrigues, "Epidemic Broadcast Trees," 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007), Beijing, China, 2007, pp. 301-310, doi: 10.1109/SRDS.2007.27.
29. Voulgaris, Spyros, Daniela Gavidia, and Maarten Van Steen. "Cyclon: Inexpensive membership management for unstructured p2p overlays." Journal of Network and systems Management 13 (2005): 197-217.
30. Ganesh, Ayalvadi J., Anne-Marie Kermarrec, and Laurent Massoulié. "Scamp: Peer-to-peer lightweight membership service for large-scale group communication." Networked Group Communication: Third International COST264 Workshop, NGC 2001 London, UK, November 7–9, 2001 Proceedings 3. Springer Berlin Heidelberg, 2001.
31. Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. 2004. Efficient Decentralized Monitoring of Safety in Distributed Systems. In Proceedings of the 26th International Conference on Software Engineering (ICSE '04). IEEE Computer Society, USA, 418–427.
32. Antoine El-Hokayem and Yliès Falcone. 2017. Monitoring decentralized specifications. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017). Association for Computing Machinery, New York, NY, USA, 125–135. https://doi.org/10.1145/3092703.3092723
33. Chris Olston, Jing Jiang, and Jennifer Widom. 2003. Adaptive filters for continuous queries over distributed data streams. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03). Association for Computing Machinery, New York, NY, USA, 563–574. https://doi.org/10.1145/872757.872825
34. J. Leitao, L. Rosa and L. Rodrigues, "Large-Scale Peer-to-Peer Autonomic Monitoring," 2008 IEEE Globecom Workshops, New Orleans, LA, USA, 2008, pp. 1-5, doi: 10.1109/GLOCOMW.2008.ECP.18.
35. J. O. Kephart and D. M. Chess, "The vision of autonomic computing," in Computer, vol. 36, no. 1, pp. 41-50, Jan. 2003, doi: 10.1109/MC.2003.1160055.
36. J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," Future of Software Engineering (FOSE '07), Minneapolis, MN, USA, 2007, pp. 259-268, doi: 10.1109/FOSE.2007.19.

37. Debanjan Ghosh, Raj Sharman, H. Raghav Rao, Shambhu Upadhyaya. Self-healing systems — survey and synthesis. Decision Support Systems, Volume 42, Issue 4, 2007, Pages 2164-2185, ISSN 0167-9236, https://doi.org/10.1016/j.dss.2006.06.011.
38. Behnaz Soltani, Venus Haghighi, Yipeng Zhou, Quan Z. Sheng, and Lina Yao. 2024. DFLStar: A Decentralized Federated Learning Framework with Self-Knowledge Distillation and Participant Selection. In Proceedings of the 33rd ACM International Conference on Information and Knowledge Management (CIKM '24). Association for Computing Machinery, New York, NY, USA, 2108–2117. https://doi.org/10.1145/3627673.3679853
39. Yacine Belal, Aurélien Bellet, Sonia Ben Mokhtar, and Vlad Nitu. 2022. PEPPER: Empowering User-Centric Recommender Systems over Gossip Learning. Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. 6, 3, Article 101 (September 2022), 27 pages. https://doi.org/10.1145/3550302
40. 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 Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11). Association for Computing Machinery, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593
41. 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