D2.3: TaRDIS architecture definition and technical specifications

Work package	WP2
Task	Task 2.3
Due date	2024-06-30
Submission date	2024-07-11
Deliverable lead	Roland Kuhn (ACT)
Version	1.0
Authors	Miroslav Popović (UNS), Ivan Kaštelan (UNS), Lidija Fodor (UNS), Dušan Jakovetić (UNS), Claudia Soares (NOVA), João Costa Seco (NOVA), João Leitão (NOVA), Diogo Jesus (NOVA), Pedro Fouto (NOVA), André Rijo (NOVA), Nuno Preguiça (NOVA), Eduardo Geraldo (NOVA), Diogo Paulico (NOVA), Tomás Galvão (NOVA), Rafael Domingues Matos (NOVA), Felipe Rossi Carmo (NOVA) Nuno Fernandes (NOVA), Diogo Ye (NOVA), Bruno Braga (NOVA), Sotirios Spantideas (NKUA), Roland Kuhn (ACT), Dimitra Tsigkari (TID), Rafael Oliveira Rodrigues (EDP), Manuel Pio Silva (EDP), Miloš Simić (UNS), Simona Prokic (UNS), Silvia Ghilezan (UNS), Ivan Prokic (UNS), Giovanni Granato (GMV), David Vázquez Enríquez (GMV), José Miguel Sánchez Martínez (GMV)
Reviewers	Ping Hou (UOX), Mário Pereira (NOVA)
Abstract	This document presents an in-depth description of the TaRDIS toolbox and its architecture and demonstrates its suitability for the implementation of the industrial use cases.
Keywords	intelligent heterogeneous swarm systems, architecture specification

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

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

Document Revision History

Version	Date	Description of change	List of contributor(s)
V0.1	2024-05-29	table of contents	Carlos Coutinho (CMS), Carlos Reis (CMS)
V0.2	2024-07-09	ready for review	*see authors list on cover page*
V1.0	2024-07-11	final version	Roland Kuhn (ACT)

Disclaimer

Funded by the European Union

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

Copyright notice

Project funded by the European Commission in the Horizon Europe Programme		
Nature of the deliverable:	**R**	
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

The TaRDIS project is creating a toolbox for implementing, analysing, and running intelligent heterogeneous swarm systems. As our requirements gathering process indicated earlier (cf. report D2.2), the range of useful swarm systems is too wide to go for a unified software stack. Instead, the TaRDIS toolbox contains tools of various kinds that can be individually composed into a suitable software stack for the programmer's task at hand. Internally, some of the tools stand for themselves while others are built upon other TaRDIS tools.

This report is the third and last of work package 2, which has defined the precise meaning of heterogeneous swarm systems for TaRDIS, and details the necessary structure of the toolbox including the requirements to be fulfilled by each kind of tool we include. The tools that have been developed so far (and some that may be added in the second half of the project) are then described and associated with their relevant set of requirements. We conclude with a small set of changes to the requirements we identified in report D2.2 filed in February 2024: these became necessary due to earlier mistakes as well as a deeper understanding gained while approaching the application of the tools to the industrial use cases that will be reported upon in D7.2 scheduled for end of September 2024.

Funded by the European Union

Table of Contents

Funded by the European Union

Funded by the European Union

LIST OF FIGURES

Funded by the European Union

LIST OF TABLES

Funded by the European Union

ABBREVIATIONS

ACT	Actyx
AI	Artificial Intelligence
API	Application Programming Interface
AST	Abstract Syntax Tree
BST	Branch Stable Time
CFSM	Communicating Finite State Machine
CRDT	Conflict-free Replicated Data Type
DCR	Dynamic Condition Response
DCS	Data-Centric synchronisation
DHT	Distributed Hash Table
DNN	Deep Neural Network
FAUNO	Federated AI Network Orchestrator
FedAvg	Federated Averaging LA
FL	Federated Learning
FLA	Federated Learning Algorithm
HLC	Hybrid Logical Clock
HTTPS	Hypertext Transfer Protocol Secure
IDE	Integrated Development Environment
IFC	Information Flow Control
IT	Information Technology
JAR	Java Archive
KD	Knowledge Distillation
KNN	K-Nearest Neighbour
LA	Learning Algorithm
LEO	Low Earth Orbit
MDP	Markov Decision Process
ML	Machine Learning
MPST	MultiParty Session Type
NN	Neural Network
NuCRDT	Non-uniform CRDT
ODTS	Orbit Determination and Time Synchronisation
P2P	Peer-To-Peer
pFedMe	Personalised FL with Moreau envelopes
PTB-FLA	Python TestBed for Federated Learning Algorithms
RL	Reinforcement Learning
SDK	Software Development Kit
SVM	Support Vector Machine
TCC	Transactional Causal Consistency
TID	Telefonica
UML	Unified Modelling Language

Funded by the European Union

1 Introduction

With this third and final report from WP2 we conclude the definition of heterogeneous swarms. In the first report, D2.1, we analysed use cases and gathered responses to a questionnaire in order to accurately grasp and describe the nature of these upcoming swarm systems, yielding a list of high-level requirements for software tools that shall ease their creation. We detailed these requirements—for each of the TaRDIS industrial use cases as well as a generic one representing prospective external users of the TaRDIS toolbox—in the second report, D2.2, starting from end-user requirements and tracing them towards low-level requirements, both in terms of technology and underlying theory.

We note that the use cases we consider cover a rather broad range of scenarios, as does the emerging field of intelligent heterogeneous swarm systems in general. Our lists therefore contain requirements whose combination is not necessarily found within a single concrete swarm project; instead, they explore a variety of possible use cases and attempt to cover as much ground as is indicated by our industrial use cases and as was discovered through our co-design process. The requirements listed in D2.2 therefore do not describe a single tool or framework, they describe a set of tools that shall be picked and composed as is warranted by the given concrete use case.

As we foresaw this already in the early stages of TaRDIS (which was possible due to the diverse set of industrial use cases), we chose to propose a solution in the shape of a toolbox, with individual tools differing in their range of applicability and their tradeoffs, but still possessing the cohesion to yield a consistent developer experience. This means that the theory behind the swarm communication primitives aligns with the required analyses to guarantee application correctness, that the chosen machine learning primitives and communication are amenable to security audits and verification, etc.

The main entry point for the application programmer will be the IDE, which corresponds to a tool rack and user guidance system. Due to this role—and the fact that the IDE is reported upon in separate deliverables D3.2, D3.4, and D3.6—we leave the IDE out of this deliverable detailing the architecture of the toolbox and of the tools within.

As one could expect, to support a wide range of complex swarm applications, the tools that TaRDIS offers in its toolbox are of different types. Some of the tools are libraries, to be used from within a specific programming language; others are separate subsystems that have their own life-cycle and can be accessed by swarm elements, like a database or middleware; yet others provide runtime support, execute declared logic fragments with particular runtime guarantees; finally, we include tools to be used solely while developing the application, to verify conformance to protocols, information security properties, and so on. How these tools are used varies greatly, which includes differences in user interface that range from command-line tools via language features to graphical editors. This is a natural result since TaRDIS covers the swarm aspects of all phases of software development, rollout, and operation. It also implies that our target audience will obtain or consume these tools via a range of delivery mechanisms, from downloading source code or application binaries to declarative project dependencies handled by a build tool.

One important aspect of this work is that the current manifestation of heterogeneous swarm tooling within our chosen programming languages represents not a limitation to that language, but offers an example on how to apply the same techniques within other languages. Our prototypes serve as de-facto standard implementation of our proposed solutions to show the viability and applicability of our ideas, but also as a clear guide for others who want to port these ideas to their own programming environments. As such, our

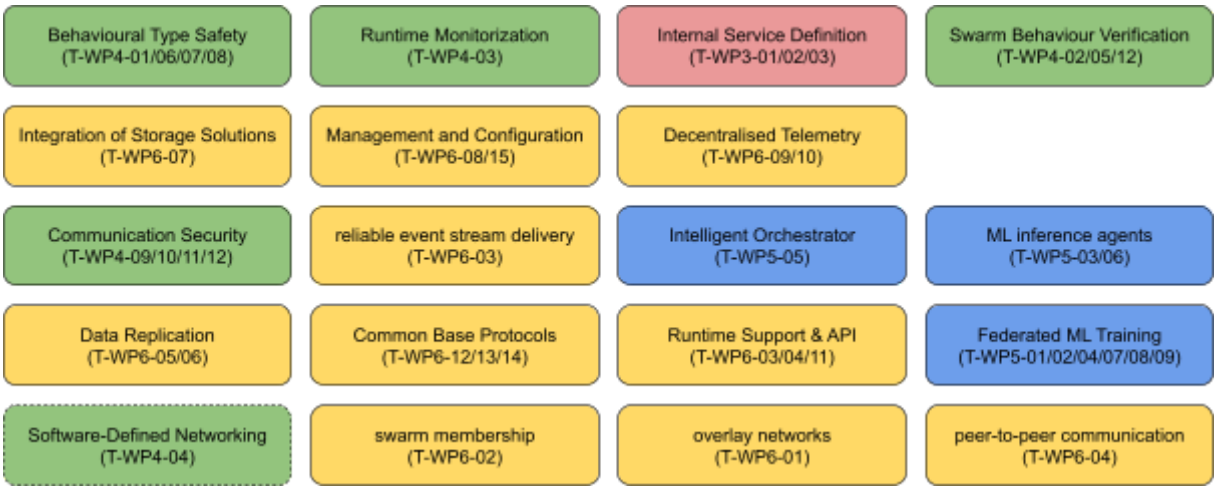
Funded by the European Union

proposed solutions can be adapted to different environments (for development and runtime) to serve an even greater target audience.

2 The TaRDIS Toolbox

In this main section we discuss the high-level architecture of the TaRDIS toolbox as it results from the co-design process followed in tasks T2.1 & T2.2. We then list all tools in the toolbox with their respective externally offered feature set and internal architecture as far as is relevant on the level of the toolbox—this includes in particular the usage of other tools or of the guarantees they offer.

2.1 Architecture Overview



TaRDIS Toolbox Architecture Overview

The above diagram shows a stratification of the TaRDIS tools, grouped by common functionality, where higher levels build upon or use guarantees of some tools on lower levels; the boxes are coloured according to their work package. It is important to note that all tools are available for use by application developers, they can even be freely combined within the same swarm application to use the right tool for any part of the job. In general, the added value and convenience increases when moving to higher levels, though.

At the lowest level we define heterogeneous swarms by employing peer-to-peer communication primitives (i.e., sending individual messages from one edge device to another via a physical transport infrastructure like wifi or ethernet). The most basic protocols using such 1-on-1 conversations are for determining the participating devices and consequently the swarm size, diameter, connectedness via a membership protocol, and the ability to form logical overlay networks on top of the physical communication topology. The shape of the latter depends on the needs of the application, hence the API allows a variety of choices in how dense and how hierarchical the overlay network will be. It should be noted that the correct function of this tool depends on the swarm size, diameter, etc. estimated by the membership service. The leftmost tool is a candidate for inclusion into the toolbox that permits software-defined networking while respecting stringent correctness constraints on the network configuration, basically filtering the swarm participants and communication topology before they reach the membership and overlay services.

On the second level, we have tools that work in the dynamically structured swarm environment provided by the first level. The first tools provide data management and replication services with different focus on strong eventual consistency among peers versus efficient but hierarchical replication that reaches up into the cloud. Then we summarise

several available protocols for common communication patterns implemented in Babel (T-WP6-04, see section 2.5.12): gossip achieves reliable information dissemination across the whole swarm without too much duplication, broadcast focuses on low-latency notification of currently connected participants, etc. The third part are supporting runtimes and their common APIs. The final piece are machine-learning algorithms that disperse intermediate training results such that ML models can be trained in a collaborative effort without sending the original training data set around—which would be prohibitive both in terms of storage size & bandwidth, and in terms of data privacy due to its provenance from human interaction.

On the third level on the right hand side we have the users of the trained ML models, namely an intelligent swarm orchestrator that controls parameters in the swarm membership and overlay network tool, as well as generic inference agents that can be used by any application code to interpret complex data sets and derive actionable information. On this level we also use the various protocols for gossip and information exchange to implement reliable event stream delivery with redundancy across both space and time—to be used by application code directly or via the tools in the top level. Finally, TaRDIS provides several powerful analyses for security properties that enforce proper usage of various kinds of communication channels.

The fourth level holds tools for the operational management of running swarm systems, including the integration with external information storage systems that are of vital importance to real-world applications of any such toolbox: data must flow between the swarm system and its business context, be that cloud-based or on-premise systems using classical IT infrastructure like database management systems, enterprise resource planning, data warehouses, archival and backup services, etc.

The topmost level of our tool stratification consists of the high-level programming language support offered to application programmers. These tools focus on delivering the increased productivity and efficiency to the developer that are afforded by the guarantees provided or enabled by the tools on the levels below. TaRDIS APIs can precisely capture how to correctly use the swarm protocols within a sufficiently powerful programming language—meaning that the language has a sufficiently expressive static type system—by using behavioural type systems. This way the programmer is alerted to mistakes without the need to write custom test code and thus guided towards correct results. The runtime monitorization tool allows a concise yet efficient declaration of conditions within the asynchronous execution of a swarm system that shall trigger alarms or any other automated response, based on complex conditions that may involve the past behaviour of several participants. In contrast to these text-based rules, the internal service definition tools allow programmers and domain experts to design swarm protocols together, based on a graphical representation that is natural to both. Finally, TaRDIS offers tools for the analysis of internal services so that the behaviour of the running swarm system matches the desired behaviour and outcomes.

In the following sections we present the detailed functionality and architecture for each of these tools, ordered by the work packages that provide them.

2.2 WP3: Programming Abstractions for the Cloud–Edge Continuum

In contrast to work packages 4–6, which provide most of the TaRDIS tools, work package 3 lays the foundation for the toolbox by defining a high-level approach towards formulating and implementing the behaviour of intelligent, heterogeneous swarm systems; it then continues to provide a set of tools that aid in applying this approach. The primary report on the groundwork has been given in deliverable D3.1. We summarise the most important findings here for the reader's convenience.

When it comes to TaRDIS applications, we differentiate between two kinds of usage of the toolbox:

1. *internal services* are declared by the programmer as part of the application logic but handed over to the TaRDIS runtime for managed execution
2. *external services* make use of TaRDIS APIs under the exclusive control of the application programmer

While the first kind affords the best correctness guarantees and convenience to the programmer, the second kind offers maximal expressiveness, flexibility, and, in particular, the ability to include TaRDIS within parts of an application that is being extended or modified in a brown-field project. It is imperative to note that within any given end-user application (i.e., one running program executed e.g., as an operating system process) TaRDIS can be used in a multitude of different ways, choosing the internal or external usage pattern as suits each specific problem case. For example, the collaboration between various swarm participants is modelled as a set of swarm protocols that are then used to derive the corresponding internal services; ML models may be declared, trained, evaluated, and used as internal services, with all swarm coordination aspects hidden from the programmer. An application may combine these internal services with direct usage of data management or machine learning APIs, and it may even employ dedicated direct communication channels between specific participants in addition to the managed information dissemination performed as part of the execution of the swarm protocols. This architecture choice resulted from the inputs collected as part of tasks T2.1 & T2.2 that have been reported upon in deliverables D2.1 & D2.2, on the definition of heterogeneous swarm systems and the identification of toolbox user requirements, respectively.

The most prominent visibility—in terms of internal services—of the approach described above manifests in the use of programming and specification languages that a developer can use via modelling tools like the WorkflowEditor (T-WP3-01), Scribble Editor (T-WP3-02), and DCR Choreography Editor (T-WP3-03). These tools allow a programmer in collaboration with the relevant experts from the application's problem domain (called *domain experts* in the following sections) to design, discuss, evolve, and implement a digital process, which we call swarm protocol at the global level and workflow at the local level (i.e., as seen from any of the participants). We describe these tools in this section because of this connection to the core programming abstractions, although all these tools also contain functionality for verifying that a swarm protocol is well-formed—this means that faithfully executing this protocol on a swarm will eventually result in the specified behaviour and outcome. Such analytic capabilities are discussed in further detail in the section devoted to tools from WP4 below.

To complement the internal services and achieve the flexibility to cover all considered use cases, all other tools presented in the sections on WP4–6 are—in addition to supporting the TaRDIS implementation itself—available to the application programmer for creating external services, with varying degrees of guarantees and developer support. Offering such a palette of choices is useful because each tool represents a specific set of trade-offs that were made between expressiveness, ease of use, code abstraction level, formal guarantees, and correctness enforced on the level of the programming language used.

2.2.1 Internal Service Definition Tools

As discussed above, the tools in this category support the definition of internal services for the purpose of running them within the TaRDIS execution engine as well as for analysis purposes. Not only does a swarm protocol require a certain discipline within its declaration to achieve the desired level of consistency during execution, it also needs to guide the implementation of the local agents by verifying their conformance to their intended role within

the protocol. This means that the programmer edits the global swarm protocol but the tool also tracks the locally projected workflows to be followed by the agents.

While there are several different approaches towards the developer experience but also a large range of tradeoffs in terms of expressivity and guaranteed consistency, the tools need to fulfil the following requirements:

T-WP3-01: WorkflowEditor
T-WP3-02: Scribble editor
T-WP3-03: DCR Choreography Editor

Internal service definition requirements

requirement ID and name	**T-WP3 -01**	**T-WP3 -02**	**T-WP3 -03**
RF-EDP-01a community acceptance and network registration	✔	✔	✔
RF-EDP-01b exchange agreement between prosumers	✔	✔	✔
RF-EDP-02 community Orchestrator for Energy Communities	✔	✔	✔
RF-TID-06 workflow orchestration	✔	✔	✔
RF-TID-08 allow hierarchies in the system	✔	✔	✔
RF-ACT-01 available swarm decision making	✔		✔
RF-ACT-02 automatic conflict resolution: eventual consensus	✔		
RF-ACT-03 replication of roles for fault tolerant responses to requests	✔		
RF-ACT-11 static analysis of swarm protocols	✔	✔	
RF-ACT-12 graphical workflow design	✔		✔
RF-WP3-MOD-01 models – graphical representation	✔		✔
RF-WP3-MOD-02 models – verification of application correctness	✔	✔	✔
RF-WP3-MOD-03 models – diverse communication topologies	✔	✔	
RF-WP3-MOD-04 models – specifying security-related requirements	✔		✔
RF-WP3-GEN-03 security and privacy	✔		✔

requirement ID and name	T-WP3-01	T-WP3-02	T-WP3-03
RNF-WP3-GEN-01 graphical representation of artefacts	✔		✔
RNF-WP3-GEN-03 ability to perform external effects in certain protocol states	✔		
RNF-WP3-GEN-04 ability to automatically execute compensating actions after conflict resolution	✔		
RF-WP3-IDE-01 graphical representation	✔		
RF-WP3-IDE-02 access to verification facilities	✔		

2.2.2 T-WP3-01 WorkflowEditor

The workflow editor supports the application developer when designing, implementing, and modifying workflows that involve multiple swarm participants. As described in D3.1, internal TaRDIS applications are formulated as workflows that are executed by the TaRDIS runtime on behalf of the local agent. Such a local workflow dictates the sequence of interactions, where the local agent can be active at certain points and passive at others (during which periods of time other agents will be active).

Implementing local workflows for all participating roles in the swarm must be done such that the overall interaction proceeds as intended. If for example a state can be reached where no participant is active while the workflow hasn't yet ended, then the application will get stuck. Such deadlock scenarios need to be avoided, which in a dynamic swarm setting means that we cannot exclude that multiple agents are active at the same time. This may lead to confusion within the swarm, so we also need to ensure that eventual consensus is reached on how the workflow has proceeded so far. Both of these properties are rather difficult to ensure in a system where the local workflows are created individually, even when that is done by the same developer—this amounts to creating a fail-safe set of governing rules for a small society, a daunting task.

The purpose of the WorkflowEditor is to make this task safe and easy. It does this by, firstly, restricting the shape of workflows using a set of analysis rules that do not permit the developer to formulate workflows that will violate eventual consensus. Secondly, it presents an overview of the global workflow (called *swarm protocol* in D3.1) instead of letting the developer create a local workflow in isolation. This global view focuses on who can do what at which state of the protocol, making it evident which participant role is responsible for pushing the workflow forward at each point; deadlock is thus impossible to construct on the global level. The local workflow will still need to be implemented for each role, but since we now do have the swarm protocol in the WorkflowEditor, the tool can ensure that the local workflow faithfully implements its role.

Within a TaRDIS application this approach is useful wherever multiple swarm participants communicate to perform a joint task that requires coordination. The corresponding interaction is typically specified as a flowchart diagram (or similar) by the respective domain experts—these are often not programmers but people specialising in the underlying processes that shall be modelled by the application. Application developers use the WorkflowEditor not only in their own implementation work but also as a communication medium with these other stakeholders.

The TaRDIS runtime depends on the integration with the Actyx middleware (see D6.1 task 6.2) since the execution of local workflows depends on the reliable and always available local event storage and swarm dissemination. It has tight interdependencies with the two libraries T-WP4-01 and T-WP4-02 described below, which therefore use the same basic tool identifier.

It should be noted that the WorkflowEditor does not yet exist as a unified tool with a graphical user interface. However, the underlying functionality is present in Actyx and the two aforementioned libraries in a first version that is fully usable.

2.2.2.1 Tool Architecture

The main part of the WorkflowEditor is an IDE plugin that offers a graphical editor for swarm protocols, with bidirectional real-time editing linking the diagram and the underlying program text in the source code file. This plugin will also run the analyses necessary for ensuring eventual consensus of the runtime execution of the modelled workflows and highlight any mistakes for the application developer to fix. It is plausible that further analyses are performed at this level, e.g., pertaining to information flow security.

Embedding the local workflow definition in a programming language (like TypeScript) requires support at compilation as well as at runtime. This support is provided by libraries that provide the functions for declaring the local workflows and detailing their required state computations. The ability to analyse a swarm protocol will also be offered as a library so that the verification may be performed as part of the automated test suite for the application.

2.2.2.2 Tool Dynamic Behaviour

The WorkflowEditor is opened by the application developer whenever they desires to work with a swarm protocol definition. Editing the graphical representation will be instantaneously reflected in the source code and vice versa; syntax errors in the code will be flagged in the code editor and may prevent the diagram from updating. Swarm protocol analysis errors are highlighted both in the diagram and in the source code.

2.2.2.3 Tool Persistence

A dedicated persistence facility for WorkflowEditor is not required since the editor works with information that is already part of the application source code and stored as such.

2.2.2.4 Tool Operational details

The WorkflowEditor is installed as any other IDE plugin, and as such it will be included in the official TaRDIS IDE bundles so that no further action is required from the application developer.

While the WorkflowEditor may work without the application having declared dependencies to the necessary runtime libraries, it does not make much sense for the developer to leave this mistake unfixed. This will usually be enforced by the typing discipline of the host programming already, e.g., TypeScript complaining that the needed type definitions cannot be found.

Opening the workflow editor requires placing the cursor within the definition of a swarm protocol data type and invoking an IDE command (which can be bound to a keyboard combination within the IDE preferences).

2.2.3 T-WP3-02 Scribble Editor

In the field of distributed and concurrent programming, achieving safety with minimal effort (i.e., through lightweight formal methods) is crucial. This is particularly important in

applications like heterogeneous swarms, where multiple agents must coordinate safely and efficiently. Session types[1] provide a typing discipline for message passing concurrency, by assigning session types to communication channels, in terms of a sequence of actions over a channel. Session types, initially only able to describe communications between two ends of a channel, are later extended to multiparty[2], giving rise to the multiparty session types (MPST) theory. The MPST typing discipline ensures that a set of well-typed communicating processes are free from deadlocks or communication mismatches. Extensions to the core theory of MPST support heterogeneous swarms: failure handling extensions include affine sessions, allowing processes to crash or fail[3], and coordinator-based failure handling techniques[4]. Other extensions enable dynamically evolving connections between swarm participants[5], dynamically sized swarm participant pools[6], and dynamically updatable multiparty session protocols[7] that support protocols with an unbounded number of fresh swarm participants, whose communication topologies are dynamically updatable. All these MPST extensions for supporting heterogeneous swarms still guarantee deadlock and communication mismatch freedom.

As an application of MPST in practice, the Scribble editor (NuScr) serves as an extensible toolchain for MPST-based multiparty protocols. This toolchain converts multiparty protocols into global types within the MPST theory. These global types are then projected into local types and further transformed into corresponding communicating finite state machines (CFSMs). Additionally, NuScr generates APIs from these CFSMs to implement endpoints in the protocol. The design of NuScr supports language-independent code generation, enabling APIs to be generated in various programming languages.

NuScr is designed for extensibility, enabling developers working on MPST to easily implement their extensions on top of NuScr. For example, extensions for failure handling, such as affine sessions and process crash management, have already been integrated into NuScr. This makes NuScr a robust platform that effectively supports the development, verification, and deployment of multiparty protocols.

2.2.3.1 Tool Architecture

Protocols in the Scribble description language are accepted by NuScr, and then converted into an MPST global type. From a global type, NuScr is able to project upon a specified participant to obtain their local type, and subsequently obtain the corresponding CFSM. Moreover, NuScr is able to generate code for implementing the participant in various programming languages, from their local type or CFSM. The codebase of NuScr can be briefly split into 4 components: **syntax**, **mpst**, **codegen**, and **utils**.

Syntax: The **syntax** component handles the syntax of the Scribble protocol description language, with the core part shown below.

[1] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In ESOP 1998.
[2] Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. Journal of the ACM 63, 1–67 (2016).
[3] Barwell, A. D., Hou, P., Yoshida, N., Zhou, F.: Designing Asynchronous Multiparty Protocols with Crash-Stop Failures. In ECOOP 2023.
[4] Viering, M., Hu, R., Eugster, P., Ziarek, L.: A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming. In OOPSLA 2021.
[5] Hu, R., Yoshida, N.: Explicit Connection Actions in Multiparty Session Types. In FASE 2017.
[6] Demangeon, R., Honda, K.: Nested Protocols in Session Types. In CONCUR 2012.
[7] Castro-Perez, D., Yoshida, N.: Dynamically Updatable Multiparty Session Protocols: Generating Concurrent Go Code from Unbounded Protocols. In ECOOP 2023.

$$
\begin{array}{rcll}
\text{Protocol Declarations } P & ::= & \texttt{global protocol } p\ (\texttt{role } \mathtt{r}_1, \cdots, \texttt{role } \mathtt{r}_n)\{G\} & \\
\text{Protocol Constructs } G & ::= & \mathtt{l}(S)\ \texttt{from } \mathtt{r}_1\ \texttt{to } \mathtt{r}_2; G' & \text{Single Message} \\
& | & \texttt{choice at } \mathtt{r}\ \{G_1\}\ \texttt{or}\ \cdots\ \texttt{or}\ \{G_n\} & \text{Branches} \\
& | & \texttt{rec } X\ \{G'\}\ |\ \texttt{continue } X & \text{Recursion / Var.} \\
& | & \texttt{end (omitted in practice)} & \text{Termination} \\
& | & \texttt{do } p(\mathtt{r}_1, \cdots, \mathtt{r}_n) & \text{Protocol Call} \\
\text{Base Types } S & ::= & \texttt{int} \mid \texttt{bool} \mid \cdots &
\end{array}
$$

A Scribble module consists of multiple protocol declarations *P*.

Multiparty Session Types: The key pipeline of handling multiparty session types is shown as below, implemented in the **mpst** component.

An input file is parsed into a **Scribble** module by the **syntax** component described in the previous paragraph. The protocols are then converted into a *global type* (defined in the **Gtype** module), which describes the overall protocol between multiple roles. A global type is *projected* into a *local type* (defined in the **Ltype** module) for a specified role, detailing the local communication behaviour. A corresponding communicating finite state machine (CFSM) (defined in the **Efsm** module) is constructed for the local type, which can then be used for API generation.

To obtain a global type, we extract it from the syntax tree of the Scribble protocol file. The **Ltype** module handles the projection of global types onto participants, converting them into projected local types. These local types can subsequently be transformed into their corresponding CFSMs, which are represented as directed graphs. Both local types and CFSMs can be used for code generation purposes.

Code Generation: The **codegen** component generates APIs for implementing distributed processes using the MPST theory. Following the top-down design methodology of MPST, processes should follow the projected local type from the prescribed global type. By the means of code generation, the processes implemented using generated APIs will be *correct by construction*.

Currently, NuScr supports code generation in OCaml, Go, and F*. Additionally, it can export the CFSM as a GraphViz Dot file, and separate code generation backends can be implemented apart from NuScr. This flexibility has proven successful in supporting code generation in TypeScript, positioning NuScr as a promising editor for supporting the languages predominantly used in TaRDIS.

API Style: The generated API separates the program logic and communication aspects of the endpoint program. Type signatures of *callback functions*, corresponding to state transitions in the CFSM, are generated for handling the program logic. These signatures are collected in a module type named **Callbacks**. Since a graph representation is used for CFSMs, the generation process involves iterating through the edges of the graph.

Utilities: The **utils** component contains miscellaneous modules fulfilling various utility functions. A few notable modules in this component, relevant to future extensions of NuScr, include:

- **Names** module defines separated namespaces for all kinds of names occurring in global and local types, e.g., payload type names, payload label names, recursion variable names, etc.
- **Err** module defines all kinds of errors that occur throughout all components.
- **Pragma** module defines language pragmas, controlling the enabled extensions.

2.2.3.2 Tool Dynamic Behaviour

Protocol Parsing and Syntax Handling:

- Parsing: NuScr reads and parses the Scribble protocol description files to convert them into an internal representation.
- Error Reporting: During parsing, NuScr detects and reports syntax errors, ensuring that protocol descriptions are correct before further processing.

Type Conversion and Projection:

- Global to Local Type Conversion: Converts global types into local types specific to each role involved in the protocol.
- Validation: Ensures that the projected local types are consistent with the global type, checking for any errors or inconsistencies.

Communicating Finite State Machines (CFSMs):

- State Representation: Local types are converted into CFSMs, representing the states and transitions that each role can undergo during communication.
- Execution Management: CFSMs manage the current state of each role and handle transitions based on received messages.

API Generation:

- API Creation: Generates APIs for each role from the CFSMs, providing a structured way to implement the protocol.
- Language Independence: Designed to support API generation for various programming languages, although the current implementation is focused on OCaml.

Runtime Execution:

- Communication Handling: At runtime, the generated APIs manage the communication between different roles, ensuring messages are sent and received as specified by the protocol.
- State Transitions: The system transitions between states based on the communication events, driven by the CFSMs.

Callback Functions:

- Separation of Concerns: The API separates communication from application. Callback functions are used to handle application-specific behaviour during state transitions.
- State-Driven Execution: Callback functions are invoked based on state transitions in the CFSMs, allowing application-specific actions to be executed at the appropriate times.

Error Handling:

- Consistency Checks: Continuously checks for consistency in message ordering and state transitions to avoid deadlocks.

Logging and Monitoring:

- Execution Logs: Logs execution details, state transitions, and communication events for debugging and monitoring purposes.
- Performance Metrics: Collects performance metrics to analyse and optimise protocol implementations.

Extensibility:

- Protocol Updates: Supports dynamic updates and modifications to protocol descriptions, allowing for flexible and adaptable protocol implementations.
- Modularity: Designed in a modular fashion, making it easy to extend and integrate with other tools or frameworks.

2.2.3.3 Tool Persistence

There are no persistent facilities required for the Scribble editor (NuScr) apart from the protocol definition files that are under the sole management of the application developer.

2.2.3.4 Tool Operational Details

NuScr can be used either as a standalone command line application, or as a library for manipulating multiparty protocols. Additionally, NuScr has a web interface[8], so that users can perform quick prototyping in browsers, saving the need for installation.

The easiest way to install is to use opam[9] via

```
opam install nuscr
```

then you can check your installation via

```
nuscr --help
```

which prints help and usage information.

2.2.4 T-WP3-03 DCR Choreography Editor

Programming distributed protocols with complex application scenarios by individually programming each node is typically a complex and error-prone task that requires dealing with several low-level details. It is a challenge to avoid execution errors, and it is a bigger challenge to avoid leaks of confidential information from the normal interactions between swarm participants. We propose an integrated approach that combines a declarative formalism based on DCR choreographies to model the global behaviour of a swarm, together with a Java communication framework, to abstract over the complex implementation of distributed protocols and systems, and an information flow control analysis on top to guarantee that the programmed behaviours do not lead to erroneous situations.

This section presents ongoing work towards designing an expressive distributed swarm programming language that can be used to design the application layer of swarms in a decentralised and dynamic context. The designed behaviours are specified using a declarative language and vetted safe by a static verification layer now under construction.

[8] https://nuscr.dev/
[9] https://opam.ocaml.org/

This language derives from prior work in ReSeda[10] and ReGrada[11] programming languages and plain Dynamic Condition Response (DCR) choreographies[12], an extension of DCR graphs with communicating participants[13]. At this stage, we use a text-based syntax for processes that specify an event-based system extended with data and time restrictions. DCR choreographies are also amenable to visual representations and editing. The current compiler implementation checks the well-formedness of a swarm specification (a choreography), which rules out the possibility of "message not understood" errors or ill-formed data expressions.

The full, compound code name of the programming language is currently (Sec)ReGraDa-IFC, extending the name of the base language ReGraDa (standing for Reactive Graph Data) with security (access control to execute events and send messages) and information flow control static checking and runtime monitoring to ensure the absence of information leaks. This confidentiality aspect is covered with the rest of the analysis capabilities as the separate tool subsection T-WP4-12.

A DCR choreography specifies the messages exchanged between participants and control flow constraints defining causality between events/messages in the system's logic. Such choreographies go beyond the traditional sequential choreography specifications (c.f. sessions[14]). It allows for a more flexible and extendable programming framework for swarms.

We provide a compiler whose source language is a DCR choreography. Its output is the projected behaviour in the form of Java code to be integrated in a fully functional communication framework. The target platform for our compiler is Babel[15], a Java communication framework for decentralised systems (T-WP6-04) that can be configured to provide the most needed runtime properties, assumed by verification tools and the application layer. Many intricate and error-prone aspects are typically inherent to designing and implementing complex distributed protocols and systems, ranging from adequately setting and timely acting upon timers to handling networking, fault-tolerance, and concurrency-control issues. Babel abstracts over the low-level details of distributed programming, such as networking, fault-tolerance, and concurrency control, and provides a high-level API for designing, implementing, and executing distributed protocols and applications. Our compiler translates a DCR choreography into a Babel protocol (a layer in a communication stack) where the local behaviour is specified, which can be executed in multiple network nodes by different participants in a swarm. Babel will also incorporate communication primitives specified and checked by related tools like (Sec)ReGraDa-IFC (T-WP4-12) that detects information leaks in swarm designs, IFChannel (T-WP4-09), which implement verified protocols, e.g., by PSPSP (T-WP4-10).

---

[10] João Costa Seco, Søren Debois, Thomas T. Hildebrandt, Tijs Slaats: RESEDA: Declaring Live Event-Driven Computations as REactive SEmi-Structured DAta. EDOC 2018: 75-84

[11] Leandro Galrinho, João Costa Seco, Søren Debois, Thomas T. Hildebrandt, Håkon Normann, Tijs Slaats: ReGraDa: Reactive Graph Data. COORDINATION 2021: 188-205

[12] Hildebrandt, T.T., López, H.A., Slaats, T.: Declarative choreographies with time and data. In: Francescomarino, C.D., Burattin, A., Janiesch, C., Sadiq, S.W. (eds.) Business Process Management Forum - BPM 2023 Forum, Utrecht, The Nether- lands, September 11-15, 2023, Proceedings. Lecture Notes in Business Informa- tion Processing, vol. 490, pp. 73–89. Springer (2023). https://doi.org/10.1007/ 978-3-031-41623-1_5, https://doi.org/10.1007/978-3-031-41623-1_5

[13] Hildebrandt, T.T., Mukkamala, R.R.: Declarative event-based workflow as dis- tributed dynamic condition response graphs. In: Honda, K., Mycroft, A. (eds.) Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010. EPTCS, vol. 69, pp. 59–73 (2010). https://doi.org/10.4204/EPTCS.69.5, https://doi.org/10.4204/EPTCS.69.5

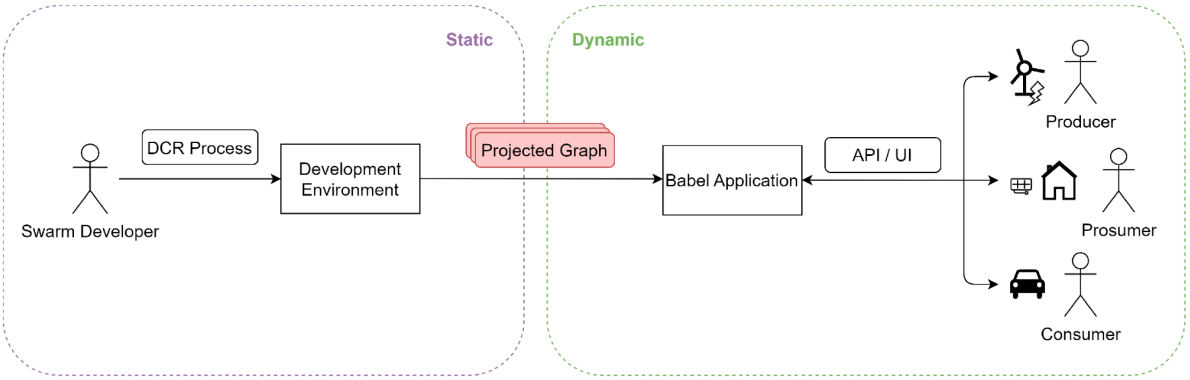
[14] Vasconcelos, V.T.: Fundamentals of session types. Inf. Comput. 217, 52–70 (2009), https://api.semanticscholar.org/CorpusID:14566897

[15] Fouto, P., Costa, P.À., Preguiça, N., Leitão, J.: Babel: A framework for developing performant and dependable distributed protocols. In: 2022 41st International Symposium on Reliable Distributed Systems (SRDS). pp. 146–155 (2022). https://doi.org/10.1109/SRDS55811.2022.00022

2.2.4.1 Tool Architecture

Swarm programming entails modelling and static checking of the swarm behaviour and enforcing the individual behaviour at runtime such that each swarm member behaves according to the global specification and security conditions. We propose a tool with a pipeline architecture, depicted in the Figure below, broadly divided into static and dynamic stages. In its present state, the editor helps build a textual representation of a DCR choreography, checking it for syntactic and typing errors. It will soon integrate (T-WP4-12) for static information flow problems. The compiler then produces code for member-specific endpoint projections defining their local behaviours. The dynamic support system is responsible for deploying and executing the Babel code for each swarm member and handling the runtime environment and Babel infrastructure, such as communication with external actors.

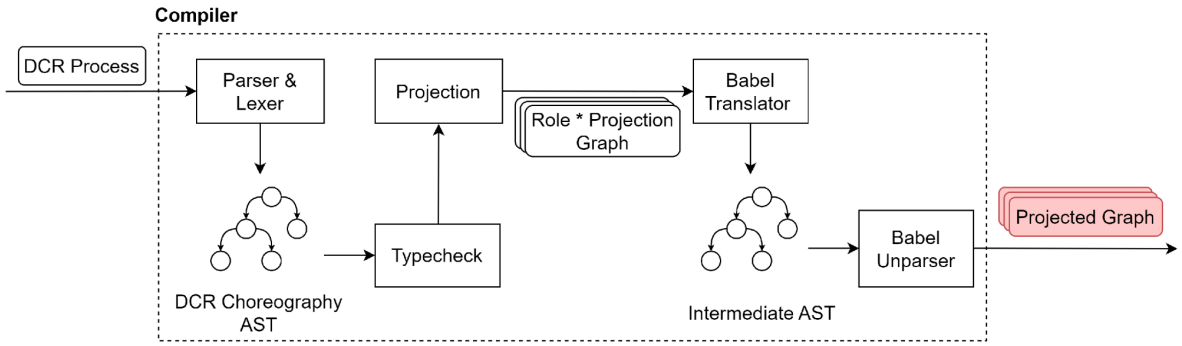
In the static stage, the Development Environment component also includes additional verification steps as part of a correct-by-design approach. At this point, type checking, projectability, and static information flow checks are performed. Similarly, the Babel Application component captures the pipeline's deployment and execution stage.



DCR Choreography Editor architecture

2.2.4.2 Tool Static Behaviour

This is the entry point to our architecture's pipeline. This stage consists of five steps, as depicted in the Figure below. The initial representation of a DCR choreography undergoes multiple transformations and verifications to produce a correct Babel program for each of the swarm members.



DCR Choreography Editor static behaviour

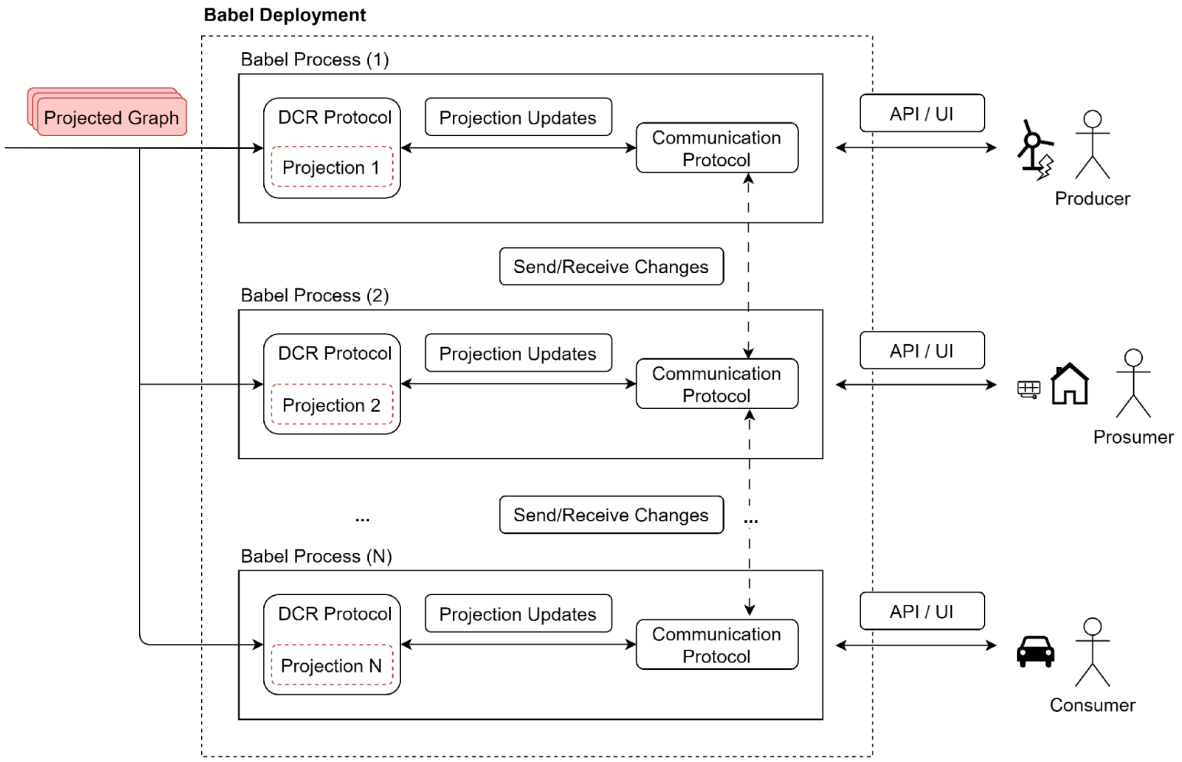
Parsing transforms high-level textual representations of DCR choreographies into abstract syntax trees (ASTs) for internal processing. The resulting ASTs are then type-checked, checked for information leaks, and checked for projectability. Data values are typed, and computation events may define simple computation expressions over data values, including input data supplied to other events. All these computations must be deemed safe at this stage. The integration of the feedback of these static verification steps in the editor provides a fluid development flow. At this stage, text-based editing and command line tools are used.

The Projection step takes as input a typed AST representing a projectable DCR choreography. It produces multiple projections, one for each participant of the swarm, of the base DCR choreography. After the projection step, the Babel translator component translates the AST of each projection to an intermediate representation, which is then unparsed to Java, encoding the local DCR graph. The next steps include statically producing the correct configuration for the communication layer.

2.2.4.3 Tool Dynamic Behaviour

Each generated Babel process enacts one DCR endpoint. As depicted in the figure below, our approach decouples DCR-endpoint process logic and distributed communication aspects into separate, orthogonal protocols. The DCR protocol component is the Babel protocol (layer) that implements the semantics of DCR graphs and is the same for all endpoints. As input, this component takes a specific projection, depending on the principal running it. The endpoint-specific behaviour is predetermined at compile-time by the projection step of the compiler and, consequently, imbued in the generated Java code. The execution of local DCR graphs is based on the state of local events and handling the constraints and side effects based on the events received and executed. A runtime monitor may be integrated into this stage if a static approach is considered too restrictive. The communication protocol component supports distribution and communication between participants, leveraging the communication abstractions provided by Babel. Decoupling communication from endpoint-specific logic makes it possible to change the communication layer to better adapt to different scenarios without changing DCR-related logic.

Funded by the European Union



DCR Choreography Editor dynamic behaviour

2.2.4.4 Tool Persistence

The DCR Choreography Editor does not require persistent facilities apart from the text-based files that contain the choreographies' textual representation.

2.2.4.5 Tool Operational Details

In its present state, the DCR Choreography Editor and associated compiler are command-line tools. Work is planned to provide visual representation and editing capabilities for DCR choreographies.

2.3 WP4: Programming Logic and Analysis Framework

At a high level, the purpose of this work package is to make the program logic (application as well as libraries and frameworks) amenable to formal analysis with the goal of ensuring correctness. This goal is not universal, though: the meaning of correctness depends on the intended use case, which cannot be understood by investigating the programming artefacts by themselves. A mathematically precise specification is required to allow a machine to verify that the program will behave as intended when it is deployed and used.

In the previous section, we presented three approaches for providing such specifications on the swarm application level. They allow the developer to declare the intended collaborative behaviour between several swarm participants such that a real-world process or workflow can be faithfully modelled in software. The process of capturing the expected program behaviour (assuming faithful execution by correct hardware and software infrastructure), comparing it to its specification derived from a swarm protocol, and checking whether the

protocol itself is well-formed is supported by *swarm behaviour verification tools* that we describe below.

The next set of tools supports the safe and correct embedding of a behaviourally specified program in a given host programming language. Each language offers its own flavour of features to aid this, from deferring all error checking to runtime (e.g., JavaScript) to proving even complex properties about the code using advanced static type systems (e.g., Scala). But even those languages with powerful built-in type checking facilities are not, in general, readily usable by the average programmer in a way that properly preserves all intended behavioural guarantees, all the way from swarm protocols down to the line-by-line program code. We, therefore, specify and include *behavioural type safety tools* in its subsection further below that enforce a behavioural typing discipline in addition to the type checking performed by the host language.

With the swarm-related program behaviour thus captured in its entirety, the third set of tools is concerned with *communication security* and is described in its subsection further below. While these tools may be used by application programmers, it should be noted that they work on a rather low level of abstraction, their regulated entities being messages and the channels they are sent over. As such, our intended usage is predominantly internal to the TaRDIS toolbox for ensuring that the higher-level services we offer to application developers, in fact do adhere to the security constraints and specifications placed on them.

Currently under investigation is the inclusion of tools for extending the reach of behavioural and security guarantees down into the logical or physical network stack that underlies all swarm communications. We thus specify *software-defined networking tools* as an optional part of the TaRDIS toolbox.

Finally, in addition to the above static analyses performed before deploying or running the code, we include tools that reach across the divide into dynamic analysis. The purpose of such *runtime monitorization tools* is to formulate rules or heuristics for certain event trace or messaging patterns that shall raise alarms, trigger mitigation actions, flag data for further processing, etc.

2.3.1 Developer Journeys

2.3.1.1 Swarm Developer Journey

The development of a swarm application using the TaRDIS toolbox starts with defining its global behaviour using one of the specification language tools available in the TaRDIS integrated environment: workflows (T-WP3-01), Scribble (T-WP3-02), or DCR choreographies (T-WP3-03). All three languages are equipped with tools to project the behaviour onto local behaviours that can be deployed in the local infrastructure of swarm participants. The help of dedicated editors and verification tools is precious to guide the developer in obtaining a well-formed behaviour definition. Verification tools range from simple type checking to ensure that there are no execution errors causing the crash of the system to more profound static analysis of the specification. Analysis ranges from communication protocol compliance to statically checking for confidentiality properties. The specification languages incorporate configuration mechanisms necessary to shape communication, data storage, and orchestration properties via language primitives, APIs, or basic configuration.

The next step in the development of the swarm is to analyse its behaviour dynamically. TaRDIS provides tools for decentralised telemetry and pattern recognition that help identify situations where reconfiguration or optimisation is needed. Intelligent orchestration tools are then used to produce the swarm's initial configuration and, based on the telemetry information, can intervene to adjust communication, workload, scheduling, etc.

2.3.1.2 Tool Developer Journey

A subset of tools in the TaRDIS toolbox is directed at the development of developer tools and infrastructural components. For instance, Babel (T-WP6-04) is being extended with cryptographic protocols, which in turn need to be certified to fulfil the properties that are assumed to be true when using it. Tools like PSPSP (T-WP4-10) target the verification of such protocols, so that when combined with the choice of protocols and the implementation of a DCR choreography in Babel, ensure that a given workflow preserves data confidentiality.

2.3.2 Swarm Behaviour Verification Tools

A swarm behaviour verification tool takes a behaviour specification as described in the WP3 section above and performs several checks to alert the programmer to certain classes of mistakes:

- **well-formedness:** the tool verifies that the specified global protocol is formed such that its distributed execution on the underlying swarm execution runtime will deliver the stated guarantees of the tool (e.g., deadlock-freedom, eventual consensus, …).
- **projection:** the tool produces a specification for each local agent that participates in the swarm protocol, checking that such specification can in fact be implemented correctly.
- **behaviour capture:** the tool extracts a model of the swarm-related behaviour of a program such that its communication actions can be automatically analysed.
- **conformance:** the tool compares the captured behaviour of a local agent to the projected behaviour for the agent's intended role in a swarm protocol and alerts the developer to all consequential deviations; it is considered good quality of service to avoid raising errors for deviations that are inconsequential during an actual swarm execution
- **documentation:** the tool provides evidence for the outcome of the analyses described above, so that application users or other developers can ascertain the correctness of the code

The table below maps the requirements from D2.2 pertaining to these functions to the tools presented later in this section:

T-WP4-02: Actyx machine-check library
T-WP4-05: Scribble
T-WP4-12: (Sec)ReGraDa-IFC

Swarm behaviour verification requirements

requirement ID and name	T-WP4-02	T-WP4-05	T-WP4-12
RF-WP3-MOD-02 verification of application correctness	✔	✔	✔
RF-WP3-MOD-03, RF-WP3-GEN-01 diverse communication topologies	✔	✔	✔
RNF-WP3-GEN-02 verification of correctness	✔	✔	✔
RNF-WP4-PROP-01 properties – communication behaviour	✔	✔	✔
RNF-WP4-VER-01 verification – communication behaviour	✔	✔	✔

requirement ID and name	T-WP4-02	T-WP4-05	T-WP4-12
RF-TID-06 workflow orchestration	✔	✔	✔
RF-ACT-01 available swarm decision making	✔		✔
RF-ACT-02 automatic conflict resolution: eventual consensus	✔		
RF-ACT-03 replication of roles for fault-tolerance	✔		
RF-ACT-11 static analysis of swarm protocols	✔	✔	✔
RF-ACT-18 swarm members may be removed and added	✔		✔
RF-WP3-GEN-04 combine devices with different capabilities/roles	✔	✔	✔

2.3.3 Behavioural Type Safety Tools

The tools specified above rely on their behaviour extraction capability to verify the conformance of each local agent to its intended swarm protocol role, which in most programming languages and for most tools is in turn driven by specific declarations added by the programmer—which are fallible, as they may not completely and accurately describe the actual behaviour of the rest of the code. This is where behavioural type checking comes in: suitably typed objects within the languages are held to a higher standard, demanding not only that inputs and outputs of all invoked operations match expectations, but also that the object is used according to a specified protocol. In other words, these tools ensure that operations are invoked in the correct order, without forgetting any actions.

When such behavioural types are directly derived from the projected swarm protocols, the implementing code achieves complete and accurate behaviour control such that mistakes are recognised and brought to the programmer's attention.

The following requirements from D2.2 are relevant for these tools:

T-WP4-01: Actyx machine-runner library
T-WP4-06: Java Typestate Checker (JaTyC)
T-WP4-07: Anticipation of Method Execution in Mixed Consistency Systems (Ant)
T-WP4-08: AtomiS – Data Centric Concurrency (an extended Java compiler)

Behavioural type safety requirements

requirement ID and name	T-WP4-01	T-WP4-06	T-WP4-07	T-WP4-08
RNF-ACT-06 programming language	✔	✔	✔	✔
RF-ACT-09 event-driven state updates	✔			

requirement ID and name	T-WP4-01	T-WP4-06	T-WP4-07	T-WP4-08
RF-ACT-10 obtain set of allowed actions for a given workflow	✔	✔		
RNF-WP3-GEN-03 ability to automatically execute compensating actions after conflict resolution	✔	✔	✔	✔
RNF-WP4-VER-06 static analysis of local agent conformance to swarm protocol role	✔	✔		

2.3.4 Communication Security Tools

Within the tools shown in the previous sections certain security assumptions are implicitly made in order to provide the services that are advertised to the user (i.e., the application author). These typically include that messages cannot be forged, the identity of the sender can be reliably established, their contents cannot be tampered with, etc. Ensuring these properties is relegated to lower level communication protocols, which is a form of *divide and conquer* approach towards making the formal analysis and proof of the offered guarantees tractable by employing this form of modularity or layering.

The tools in this section rely on very simple low-level primitives—point-to-point channels and the messages they carry—to build secure communication primitives on top. While the actual implementation of higher-level protocols falls into the responsibility of WP6, these tools are needed to verify that the WP6 implementations are correct.

A security tool is a helper that is external to the program under study and allows the developer to reason about its security properties with mathematical precision and accuracy.

T-WP4-09: IFChannel
T-WP4-10: PSPSP
T-WP4-11: CryptoChoreo
T-WP4-12: (Sec)ReGraDa-IFC

Communication security requirements

requirement ID and name	T-WP4-09	T-WP4-10	T-WP4-11	T-WP4-12
RNF-EDP-02 Security and Privacy	✔	✔	✔	✔
RF-TID-01 secure communications between entities	✔	✔	✔	✔
RF-TID-02 secure communications between applications	✔	✔	✔	✔
RF-TID-03 privacy of FL clients	✔			✔
RF-ACT-21 cryptographic key management		✔	✔	

requirement ID and name	T-WP4-09	T-WP4-10	T-WP4-11	T-WP4-12
RF-ACT-22 multiple swarms run on the same network infrastructure without interference		✔		
RF-ACT-23 trusted swarm membership with easy joining		✔	✔	
RF-WP3-GEN-03 security and privacy	✔	✔	✔	✔
RNF-WP4-PROP-03 properties – security and privacy	✔	✔	✔	✔
RNF-WP4-VER-03 verification – information flow	✔			✔
RF-WP6-G-01 management of cryptographic material by participant		✔	✔	
RF-WP6-MA-04 authenticated decentralised membership abstractions		✔	✔	
RF-WP6-MA-08 isolation between different membership abstractions		✔		✔
RF-WP6-CP-16 point-to-point secure communication primitives		✔	✔	✔
RF-WP6-CP-17 point-to-multipoint secure comm. primitives		✔	✔	

2.3.5 Software-Defined Networking Tools (under consideration)

Even lower-level than the aforementioned security tools is the functionality of the communication network itself. In recent years, network infrastructure has increasingly become logical rather than physical, meaning that the function of a switch or router is no longer determined by its physical wiring but by the internal pathways configured via software. The tools in this category have been developed thanks to this evolution, but we intend to adapt them to the swarm context, where all infrastructure is decentralised. Instead of configuring pathways within a router, software-defined networking in TaRDIS means the selective permission of messaging pathways between swarm participants. This extends the reach of security policies further down the stack to allow constraints such as allowing unauthorised (joining) swarm participants only access to selected endpoints on selected devices.

A software-defined networking tool takes the shape of a library for interacting with the low-level network infrastructure that is parameterised with a set of security and safety policies. The library will prevent violating instructions from being performed. It will also employ the host programming language's type system to recognise violations during static program analysis where possible.

Funded by the European Union

T-WP4-04: P4R-Type

Software-defined networking requirements

requirement ID and name	T-WP4-04
RF-ACT-22 multiple swarms run on the same network infrastructure without interference	✔
RF-ACT-23 trusted swarm membership with easy joining	✔
RF-WP6-MA-08 isolation between different membership abstractions	✔
RF-WP6-CP-16 point-to-point secure communication primitives	✔
RF-WP6-CP-16 point-to-multipoint secure comm. primitives	✔

2.3.6 Runtime Monitorization Tools

The final set of tools straddles the divide between static and dynamic analysis by allowing the static declaration of policies to be applied to a running swarm system. This is useful when not all formally permissible swarm protocol executions are equally desirable, consider for example a workflow in a factory that may fail. While failure is an acceptable outcome for the static analysis, the application designer may want to trigger some action in this case because the frequency at which failures occur is a prominent target for optimisation while striving to keep the factory profitable while offering competitive pricing.

A monitorization tool takes the shape of a development library or middleware that is parameterised by the programmer to observe a stream of events generated by the swarm and react to certain patterns. These can be specific event sequences, including constraints on the time intervals between them, or the frequency with which some events occur. The programmer can specify which action to take should any such pattern be recognised in the swarm system.

T-WP4-03: Formalised & Optimised Library for Join Pattern Matching

Runtime monitorization requirements

requirement ID and name	T-WP4-03
RF-WP3-GEN-01 diverse communication topologies	✔
RF-WP3-GEN-02 logging and monitoring of application activity and status	✔
RF-WP3-GEN-05 reconfiguration upon detection of relevant events	✔
RF-ACT-24 efficient event trace pattern matching	✔

2.3.7 T-WP4-01 Actyx machine-runner library

This tool accompanies the WorkflowEditor (T-WP3-01), serving a triple purpose:

1. it offers an API for declaring local workflow implementations,
2. provides the execution engine for local workflows, and
3. ensures correct interaction with running local workflows through extensive static type checking.

machine-runner is a TypeScript library available from the npmjs.org global repository.

2.3.7.1 Tool Architecture

The first aspect implements *behaviour capture* in the sense of a swarm behaviour verification tool. To this end machine-runner offers an internal DSL hosted in the TypeScript language with primitives for declaring workflow states, their payload data types, offered commands, and event-driven state transitions. The DSL captures the structure of the workflow such that it can be analysed for the purpose of conformance to a swarm protocol role, but it also captures on the level of the TypeScript static type system information on which payload data and commands are available in each workflow state. This is important for the third aspect discussed below.

The second function is performed by tracking the workflow execution based on the emitted events, both from the local node and other swarm participants. The handling of event streams is done by the Actyx middleware (T-WP6-03), whose responsibility it is to persist and disseminate events between all workflow participants.

The third aspect uses the static type information obtained from the behaviour capture DSL and leverages the powerful TypeScript type system to verify that the application code correctly handles each workflow state. The main language construct for this purpose is the reaction to the stream of state updates from an asynchronous `` `for await` `` loop. Within the loop body the programmer examines the current state using a `if else` ladder, at the end of which they assert the handling of any possible state such that omissions will be flagged as compilation errors. The code for handling each state then is verified to expect the correct state payload and only invoke commands that are permitted within this state. Once the events emitted via the command have been persisted in Actyx and returned with full metadata to the application, the aforementioned execution engine computes the next state based on the declared local workflow and the `` `for await` `` loop is then triggered again.

2.3.7.2 Tool Operational Details

As with any other library, it is installed by invoking

```
npm install @actyx/machine-runner
```

within the project directory. Its APIs for declaring local workflows and executing them are documented using JsDoc comments, which is shown with all modern IDEs that support content assist.

Running an application using this tool requires the Actyx SDK library (@actyx/sdk) and a locally running Actyx middleware process. All these software packages are available under the Apache 2 open-source licence.

2.3.8 T-WP4-02 Actyx machine-check library

This TypeScript library is a companion tool to the machine-runner library (T-WP4-01). It implements the verification of swarm behaviour captured as swarm protocols using the

WorkflowEditor (T-WP3-01) and implemented using the machine-runner DSL based on the underlying theory developed within the TaRDIS project[16].

2.3.8.1 Tool Architecture

The library itself is written in the Rust programming language to leverage its strong correctness and performance guarantees. It is then compiled using the WASM toolchain and offered as a Javascript library with exhaustive TypeScript type declarations. The two main functions are `checkSwarmProtocol` and `checkProjection` and take the swarm protocol definition in JSON format. Internally, the verification of a swarm protocol consists of nested traversals of the protocol's state machine graph; checking a projection means computing the intended role's projection and comparing this graph to the one captured by the machine-runner DSL. In either case, the result is an array of error messages, which is handed back into the Javascript domain as the function return value.

2.3.8.2 Tool Operational Details

As with any other library, it is installed by invoking

```
npm install @actyx/machine-check
```

within the project directory. Its functions are intended to be used before the application is deployed. The best way is to include invocations of `checkSwarmProtocol` and `checkProjection` in the unit tests, thus ensuring that the protocol is well-formed and the local workflows faithfully implement their assigned roles before the application is run.

2.3.9 T-WP4-03 Formalised & Optimised Library for Join Pattern Matching

Join patterns provide a promising approach to the development of concurrent and distributed applications where different components may need to interact using complex message combinations and conditions. A join pattern (with conditional guard) is reminiscent of a clause in a typical pattern matching construct: it has the form "J *if* $\gamma \Rightarrow C$" — where J is a message pattern describing a combination of incoming messages and binding zero or more variables, and γ is a guard, i.e., a boolean expression that may use the variables bound in J. A program using join patterns can wait until a desired combination of messages arrives (in any order); when some of the incoming messages are matched by the message pattern J and (their payloads) satisfy the guard γ, the code C is executed.

This tool (developed by DTU) is a ready-to-use library for the Scala 3 programming language allowing developers to write programs based on join patterns using an intuitive API. The library implements different matching strategies with different performance characteristics and trade-offs, depending on the volume and shape of the incoming messages, and on the shape of the patterns being matched.

This tool will be used in the Actyx use case for writing a monitoring program that observes the flow of messages describing ongoing activity on a factory shop floor, match some combinations of messages, and maintain statistics and report quality of service (QoS) issues.

This tool has no dependencies with other TaRDIS tools but works very well together with the reliable event stream replication of T-WP6-03.

[16] Roland Kuhn, Hernán Melgratti, Emilio Tuosto: Behavioural Types for Local-First Software. ECOOP 2023, https://doi.org/10.4230/LIPIcs.ECOOP.2023.15

2.3.9.1 Tool Operational details

The tool is a regular Scala 3 library that can be imported and used in a standard way. The following image shows an initial fragment of the monitoring program for the Actyx use case (mentioned above): the various 'case's of the pattern matching describe various combinations of messages, and the code that is executed when such messages are observed.

```
def monitor() = Actor[Event, Unit] {
    receive { (self: ActorRef[Event]) => {
        case ( Fault(_, fid1, _, ts1),
               Fix(_, fid2, ts2) ) if fid1 == fid2 =>
            updateMaintenanceStats(ts1, ts2)
            Continue

        case ( Fault(mid, fid1, description, ts1),
               Fault(_, fid2, _, ts2),
               Fix(_, fid3, ts3) ) if fid2 == fid3 && ts2 > ts1 + TEN_MIN =>
            updateMaintenanceStats(ts2, ts3)
            log(s"Fault ${fid1} ignored for ${(ts2 - ts1) / ONE_MIN} minutes")
            self ! DelayedFault(mid, fid1, description, ts1) // For later processing
            Continue

        case ( DelayedFault(_, fid1, _, ts1),
               Fix(_, fid2, ts2) ) if fid1 == fid2 =>
            updateMaintenanceStats(ts1, ts2)
            Continue

        case Shutdown() => Stop()
    } }
}
```

The code above (which uses a rather intuitive API) is internally rewritten by the library (using the Scala 3 macro system) to realise the corresponding message pattern matching logic which is non-trivial and includes various optimisations.

2.3.10 T-WP4-04 P4R-Type

Many modern network switches and routers provide Software-Defined Networking (SDN) capabilities, which allow for writing control programs that can define and alter the network's packet processing rules.

P4R-Type is a Scala 3 library (developed by DTU) for developing SDN control programs, adhering to the P4Runtime standard.[17] If a control program using P4R-Type can be compiled, then all its operations that may modify the P4 network configuration are guaranteed to be compliant with the network specification (i.e., all updates will respect the packet processing tables, allowed actions, and action parameters).

[17] https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

P4R-Type is being investigated as a possible component of the TaRDIS toolbox. Its intended application would be to regulate swarm membership at the network level, using SDN to allow new swarm participants only after a successful authentication.

This tool has no dependencies with other TaRDIS tools.

2.3.10.1 Tool Operational details

P4R-Type consists of two main components: a command-line program that automatically generates Scala 3 data types from P4 SDN configuration metadata, and a Scala 3 library to write programs that use such types.

The typical workflow for P4R-Type is the following:

- A network configuration is deployed on one or more routers, using standard P4 tooling
- As a result of the previous step, a series of P4Info metadata files are produced (as part of the standard P4 workflow). Each P4Info file describes the network configuration of a P4-enabled device on a network.
- Given a P4Info file called e.g., 'router1.p4info', a software developer can invoke the P4R-Type data type generation tools as follows:

  ```
  ./p4rtypegen router1.p4info config1
  ```

  This produces a file called 'config1.scala' containing a package (of the same name) with a set of data types — which correspond to the network configuration in 'router1.p4info'. This step can be repeated for multiple P4-enabled network devices.
- Then, the software developer can start writing Scala code by using the P4R-Type library, and importing the types in the file 'config1.scala' above.

The following image shows a sample program that attempts to update the configuration of two network switches (described in the packages 'config1' and 'config2', both generated using the command-line tool 'p4rtypegen' mentioned above). The line with an error highlights an invalid update, that is detected at compile-time.

```
@main def forward_c1() =
  val s1 = config1.connect(0, "127.0.0.1", 50051)
  val s2 = config1.connect(1, "127.0.0.1", 50052)

  insert(s1, TableEntry(
    "Process.ipv4_lpm",
    Some("hdr.ipv4.dstAddr", LPM(bytes(10,0,1,1), 32)),
    "Process.ipv4_forward",
    (("dstAddr", bytes(8,0,0,0,1,17)), ("port", bytes(1))), 1
  ))

  insert(s1, TableEntry(
    "Process.ipv4_lpm",
    Some("hdr.ipv4.dstAddr", LPM(bytes(10,0,1,1), 32)),
    "NoAction", // <-- This action only exists in Process.firewall
    (), 1
  ))
```

2.3.11 T-WP4-05 Scribble

In today's realm of distributed and concurrent programming, the quest for safety with minimal effort, often through lightweight formal methods, has become a significant focus. Session types, particularly multiparty session types (MPST), provide a typing framework for message passing concurrency by assigning types to communication channels based on sequences of actions. This framework ensures that sets of well-typed communicating processes remain free from deadlocks or communication mismatches.

Scribble (NuScr), an extensible toolchain for MPST, has three main aspects:

1. a language for specifying global interactions;
2. a tool to manipulate specifications and generate implementable APIs; and
3. a theory supporting the safety guarantees.

This tool can be used either as a standalone command-line application or as an OCaml library for handling multiparty protocols. Additionally, NuScr offers a web interface[18], enabling users to conduct prototyping directly without the need for installation.

In contrast to the swarm protocols designed and implemented using the WorkflowEditor (T-WP3-01), Scribble enforces a consistent communication discipline—every participant action is permitted only when it is consistently possible, no actions need to be reverted in case of intermittent communication failures. This stricter discipline is sometimes necessary for more delicate workflow problems, but it also places tighter restrictions on which workflows can be modelled. This difference in trade-offs makes Scribble a complementary tool to the WorkflowEditor, even though both aim to achieve declarative behaviour design for a heterogeneous swarm system.

This tool has no dependencies with other TaRDIS tools.

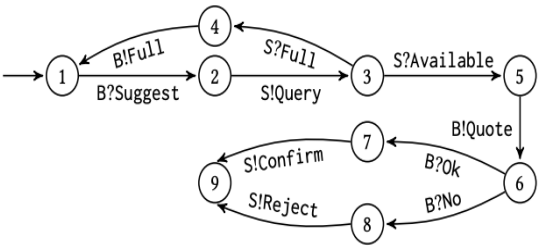
2.3.11.1 Tool Operational details

NuScr accepts protocols described in the Scribble language, as exemplified in the figure below, and subsequently converts them into an MPST global type.

```
1  global protocol TravelAgency(role A, role B, role S)
2  { Suggest(string) from B to A;//friend suggests place
3    Query(string) from A to S;
4    choice at S
5       { Available(number) from S to A;
6         Quote(number) from A to B;//check price with friend
7         choice at B
8             { OK(number) from B to A;
9               Confirm(credentials) from A to S; }
10        or  { No() from B to A;
11              Reject() from A to S; } }
12   or { Full() from S to A; Full() from A to B;
13        do TravelAgency(A, B, S); } }
```

From a global type, NuScr can project upon a specified participant to obtain their local type and subsequently derive the corresponding communicating finite state machine (CFSM), as shown below. Additionally, NuScr can generate code to implement the participant in various programming languages, based on their local type or CFSM.

[18] https://nuscr.dev/



NuScr example protocol state machine

2.3.12 T-WP4-06 Java Typestate Checker (JaTyC)

In software systems, resources are stateful and operations performed on them may depend on properties of the state. For instance, one cannot pop from an empty buffer or withdraw from an account without sufficient balance. These properties about state, when declared in the code implementing the system, are called *typestates*. A simple way of representing them is like finite automata, where transitions from a certain state correspond to operations that can be performed safely (i.e., without eventually crashing the application or leading it to an incoherent state).

The key idea of **JaTyC** is to associate a typestate annotation with every stateful class, declaring the object's states, the methods that can be safely called in each state, and the states resulting from the calls. The tool statically verifies that when a Java program runs: sequences of method calls obey to object's protocols; objects' protocols are completed; null-pointer exceptions are not raised; subclasses' instances respect the protocol of their superclasses. Moreover, it supports protocols to be associated with classes from the standard Java library or from third-party libraries and supports "droppable" states, which allow one to specify states in which an object may be "dropped" (i.e., stop being used) without having to reach the final state. The latest version adds support for subtyping: you may have a class with a protocol that extends another class with another protocol and the tool will ensure that the first protocol is a subtype of the second protocol.One can also create a class with protocol that extends a class without protocol. In the class without protocol, all methods are available to be called and remain so in the subclass. Then in the subclass, one can add new methods and restrict their use by only allowing them in certain states. More information can be found in the tool repository[19].

This tool has no dependencies with other TaRDIS tools. It may be particularly useful to implement memory-safe code by construction building on the TaRDIS APIs. Moreover, these APIs can be documented with typestates that describe them from a behavioural perspective.

2.3.12.1 Tool Operational details

JaTyC is a plugin for the Checker Framework[20]. It is a purely transparent checker, i.e., does not modify the baseline Java compilation. So, a simple way of using the tool is to incorporate it in the normal compilation process. Thus, to use **JaTyC** the typical workflow is the following[21]:

1. download and install checker-framework;
2. get `jatyc.jar`

[19] https://github.com/jdmota/java-typestate-checker
[20] https://checkerframework.org/
[21] https://github.com/jdmota/java-typestate-checker?tab=readme-ov-file#installation

3. run in a terminal the command java -jar on the folder where your typestate annotated source Java files are, with the appropriate path containing the files obtained in steps 1 and 2.
4. if the code has no typical Java compile errors, then the typestate annotations are checked; the output of **JaTyC** is either OK or a list of errors as below.

Consider an Iterator over some collection that has already visited all elements of the collection. Calling method next() in “plain” Java may produce a null-pointer exception at runtime; with **JaTyC** one gets a compile-time error:

```
Main.java:6: error: Cannot call [next] on State{JavaIterator, end}
    iterator.next();
                ^
```

2.3.13 T-WP4-07 Anticipation of Method Execution in Mixed Consistency Systems

Distributed applications (widely common these days) need to replicate data to make it available. Eventually, possible conflicts must be solved. A typical example is a shared set: inserts can always happen, as sets do not have repeated elements (although the local view of the set may be outdated), but removals require causal and/or eventual "coordination" (if one cannot remove a non-existing value, as in some contexts, this can block or crash the device).

Ant is an approach to determine statically operations that can safely commute with other operations in replicas of a distributed system. The information is used to allow a run-time system to anticipate calls to commutable operations. The theory behind is described in papers published recently[22][23].

The aim is to reduce the programmer’s effort by only requiring simple and intuitive annotations at data declaration. The goal is to use the annotations to automatically identify all accesses to replicated data; operations accessing such data are either conflict-free with other operations or may require coordination.

The **Ant** approach as been implemented in a tool - **REPL**[24] that performs compile-time commutativity analysis for the Java language, computing the commutativity of pairwise method calls from the input given at data declaration, parameters’ values and fields’ states.

This tool has no dependencies with other TaRDIS tools. The approach and/or the tool can be particularly useful to control at runtime the execution of operations in the swarm replicas, ensuring eventual data-consistency.

2.3.13.1 Tool Operational details

Given a Java class with annotations (a simple example is below), stating field invariants, method pre-conditions, and crucially, which operations require coordination, the tool produces a map of functions for each method call pair with the corresponding commutativity conditions, which can be used at runtime to determine if method calls require some form of consistency (e.g., strong, eventual, causal, etc) or can execute without global coordination.

[22] https://doi.org/10.1145/3555776.3577725
[23] https://arxiv.org/abs/2212.14651
[24] http://hdl.handle.net/10362/164248

```java
public class Account {
    @Repl @Invariant("balance >= 0") private int balance;

    @MethodCondition(expression = "param0 > 0")
    public void deposit(int amount) {
        if (amount > 0) this.balance += amount;
    }

    @MethodCondition(expression = "param0 > 0")
    public void withdraw(int amount) { this.balance -= amount; }

    @MethodCondition(expression = "param1 > 0")
    public void transfer(Account to, int amount) {
        withdraw(amount);
        to.deposit(amount);
    }

    @MethodCondition(expression = "param0 > 0")
    public void accrueInterest(int interestRate) {
        if (interestRate > 0) this.balance = this.balance + this.balance * interestRate;
    }

    public int getBalance() { return this.balance; }

    @ConsistentRead
    public int getConsistent_Balance() { return this.balance; }
}
```

The map produced by the tool is the following.

	$accrueInterest(y_1)$	$deposit(y_1)$	$getBalance()$	$getConsistent\_Balance()$	$transfer(y_1, y_2)$	$withdraw(y_1)$
$accrueInterest(x_1)$	$true$	$false$	$true$	$(not\ (>\ balance\ 0))$	$false$	$false$
$deposit(x_1)$	$false$	$true$	$true$	$false$	$(not\ (>\ y_2\ balance))$	$(not\ (>\ y_1\ balance))$
$getBalance()$	$true$	$true$	$true$	$true$	$true$	$true$
$getConsistent\_Balance()$	$(not\ (>\ balance\ 0))$	$false$	$true$	$true$	$false$	$false$
$transfer(x_1, x_2)$	$false$	$true$	$true$	$false$	...	$(not\ (>\ y_1\ balance))$
$withdraw(x_1)$	$false$	$true$	$true$	$false$	$(not\ (>\ y_2\ balance))$	...

TaRDIS could have a library to manage operations at run-time, consulting the map and either allowing immediately an operation to be performed or to trigger an alarm/signal notifying for the need of coordination. Orchestrated use cases may force all replicas to apply the operation; Choreographic ones would mark a point to return to when an irreconcilable divergent state is later identified).

2.3.14 T-WP4-08 AtomiS - Data Centric Concurrency (an extended Java Compiler)

Data-Centric synchronisation (DCS) shifts the reasoning about concurrency restrictions from control structures to data declaration. It is a high-level declarative approach that abstracts away from the actual concurrency control mechanism(s) in use. AtomiS[25] requires only qualifying types of parameters and return values in interface definitions, and of fields in class definitions. The latter may also be abstracted away in type parameters, rendering class implementations virtually annotation-free. From this high level specification, a static analysis

[25] Hervé Paulino, Ana Almeida Matos, Jan Cederquist, Marco Giunti, João Matos, António Ravara: *AtomiS: Data-Centric Synchronization Made Practical*. Proc. ACM Program. Lang. 7 (OOPSLA2): 116-145 (2023). https://dl.acm.org/doi/10.1145/3622801

infers the atomicity constraints that are local to each method, considering only the method variants that are consistent with the specification, and performs code generation for all valid variants of each method. The generated code is then the target for automatic injection of concurrency control primitives that are responsible for ensuring the absence of data-races, atomicity-violations and deadlocks. In short, AtomiS is used to mark resources which need to be accessed in mutual exclusion; a type-checking and inference system ensures race freedom.

This tool has no dependencies with other TaRDIS tools, but can be used in subsequent releases of some of them. When developing concurrent applications that share resources, or in particular in the case of orchestrated TaRDIS tools like FAUNO (WP5-05), Babel (WP6-04), or Distributed Management of Configuration based on Namespaces (WP6-08), a critical aspect is the identification of the right concurrency control features and where to place them. AtomiS can thus be used to optimise and produce by-construction thread-safe versions of these tools.

2.3.14.1 Tool Operational details

Given a Java class with annotations (a simple example of annotations added to well-known APIs is below), stating which resources (fields of parameters) should be accessed in mutual exclusion, the tool checks if the annotations are consistent and if they are, produce the necessary concurrency control to run safely the code.

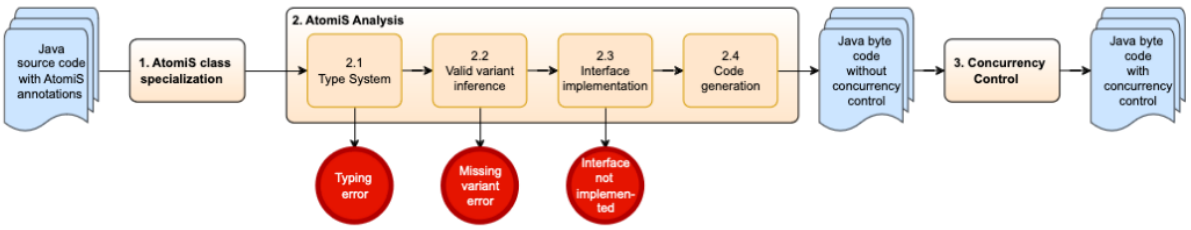
```
interface ListAtomic {
 void add(@Atomic Object element);
 @Atomic Object get(int pos);
 boolean containsAll(@All ListAtomic other);
}

class NodeOfAtomic {
 @Atomic Object value;
 NodeOfAtomic next, prev;
}
```

```
class ConcurrentListOfAtomic implements
      ListOfAtomic {
 @Atomic NodeOfAtomic head, tail;

 void add(Object element) { ... }
 ...
}
```

The figure below describes the approach: the class specialisations (automatically generated) are versions for each combination of {atomic-qualified, not atomic-qualified} for each type parameter; the analysis (and synthesis) type-check (using a bi-direccional approach) atomicity-related overloaded versions of each method; if all goes well, the atomicity annotations are stripped from the original code and concurrency control (thread-safe by construction) is included in the original and in the generated code. A standard Java compiler can then take over.



AtomiS operational pipeline

2.3.15 T-WP4-09 IFChannel

The IFChannel tool is a plugin for information flow analysis where data is transmitted over secure channels, but these channels are implemented as encrypted messages over a public network. The main use of this tool for now is within the (Sec)ReGraDa tool (see below T-WP4-12); we plan to make it also available as part of a general purpose information flow analysis tool.

The purpose is that in a DCR choreography, where part of the data is labelled as confidential within a group of people, one can use the (Sec)ReGraDa and IFChannel to verify that there is no information flow to participants outside the group. The IFChannel plugin here ensures that this result also holds when the channel is implemented by cryptography and the exchange of encrypted messages is observable by an intruder; in this case classical information flow does not hold, but a weakened version as proved in a work of WP4 (cited below).

All TaRDIS use cases that can use DCR choreographies can use the benefits of this approach; namely the EDP use case.

The input is a DCR choreography and the output is either a confirmation that information flow conforms to a given policy, or a point in the DCR graph that violates the information flow policy, i.e., where information can flow to unauthorised participants. The combination of IFC statically verified in DCR choreographies and IFChannel lifts some of the assumptions about communication of the first approach and adapts it to a scenario where communications can be intercepted but not understood.

2.3.15.1 Tool Operational details

The tool is currently conceived as part of the (Sec)ReGraDa tool (T-WP4-12); the work that has been done so far is the security proof of the concept; this is part of a submitted paper for a conference[26] and summarised in deliverable D4.2 of WP4. There is no installation so far.

2.3.16 T-WP4-10 PSPSP

PSPSP is protocol verification tool integrated into and verified by Isabelle/HOL. It can automatically verify a class of protocols with mutable long-term state. The protocols must be written in the transaction-based specification language of PSPSP called *trac*.

For TaRDIS, we do not envision PSPSP as an integrated tool, available for the average TaRDIS user. Rather, PSPSP should be used to verify the built-in channels and protocols offered as a part of the TaRDIS toolkit. A primary concern is to check that these channels indeed give the guarantees we rely on for the soundness of IFChannel.

A possible extension would be to allow expert TaRDIS users to implement their own channels and protocols, if they have special requirements not covered by out-of-the-box TaRDIS. We might then allow these experts to verify their own implementations in PSPSP and automatically check that they compose properly with the existing library of protocols, and that the new channels give sufficient protections for IFChannel.

The output of the tool is a proof certificate in Isabelle/HOL that the protocol satisfies its security goals (within the Dolev-Yao intruder model). This also involves proving a property of the implementation of message formats: that they are unique and pairwise disjoint. This needs to be done manually, while all other proofs can be derived automatically in PSPSP (if they hold).

[26] Simon Tobias Lund and Sebastian Mödersheim, Dolev-Yao Information Flow, submitted, 2024

We have established the security of several known protocols with PSPSP, for instance TLS. We are currently checking which security protocols shall be implemented within WP6 (i.e., connected to the Babel (T-WP6-04) framework) and those will then be verified with PSPSP. Then they should be used in all use cases that require secure transmissions, e.g., EDP.

2.3.17 T-WP4-11 CryptoChoreo

This is a projected tool to improve the support for specifying and verifying the cryptographic communication protocols that implement the channels provided by the TaRDIS API. Recall that the information flow analysis (T-WP4-09 and T-WP4-12) assume that communication between actors needs to be over secure channels when confidentiality and integrity of data is part of the policy. Similarly we also rely on security protocols when, for instance, keys are updated, or group memberships change. Currently, we use the PSPSP tool (T-WP4-10) to verify such protocols, but this requires a specification in the low-level input language *trac* of PSPSP. This is essentially the local behaviour of each participant, broken down into single transactions. The CryptoChoreo tool will allow an alternative way to specify protocols as a choreography, i.e., a collaborative process of the actors. We note that, like PSPSP, this tool is planned for the internal use of the TaRDIS project when implementing channels, as well as potentially for TaRDIS power users who wish to extend the library with further verified channels.

The input language for CryptoChoreo is a specialised dialect of standard choreography languages, i.e., where instead of events we have cryptographic messages with encryption operations, hashes and the like. Moreover, one needs to specify which messages each actor initially knows at the beginning of the Choreography, as well as a number of security goals. The CryptoChoreo tool shall analyse this specification for how every actor locally should execute the protocol: how they process incoming messages (decryption, and checks that they make) and how to produce outgoing messages. In this analysis the protocol may be recognized as *un-executable*, i.e., at least one actor cannot perform a step of the protocol (e.g., they are supposed to decipher a message for which they do not have the decryption key) or the next step of an actor cannot be properly determined because there is a non-deterministic fork in the choreography and the actor cannot determine which branch is the case.

In case of an in-executable protocol, the designer of the channel needs to change the protocol: it cannot be implemented in this way. In all other cases, the CryptoChoreo tool will produce a set of local execution of the different actors and we plan to have output for PSPSP (and other protocol verification tools in the future). In this way, the verified protocol can be directly fed into verification tools. This should either result in a positive verification result - i.e., the protocol indeed satisfies its security goals - or a failure. The failure is either due to a possible attack against the protocol (a Dolev-Yao intruder can break a security goal) or due to the over-approximation that PSPSP (and other tools) perform as part of their abstract interpretation. For this case of failed verification we plan to devise heuristics to translate the derivation of the special *attack* symbol in the abstract model back to a corresponding trace of the choreography. This “back-translation” can only be a heuristic as the general problem is undecidable, and the abstract interpretation techniques are a way to side-step this undecidability.

Last but not least, it is possible to derive a “skeleton implementation” from a choreography, since the translation identifies the sequence of cryptographic operations that an actor has to perform. Thus, when filling concrete cryptographic algorithms with their implementation details like key sizes, we can automatically produce programs that implement the choreography from the point of view of any actor.

So far, the groundwork for this tool has been laid, and we envision an implementation in the second half of the project to be available for TaRDIS “power users” to verify new channel implementations.

2.3.18 T-WP4-12 (Sec)ReGraDa-IFC and DCR Choreographies

This tool complements T-WP3-03, the editor and compiler of DCR choreographies to Babel infrastructures. Programming distributed protocols with complex application scenarios by individually programming each node is a complex and error-prone task, where program logic errors, simple execution errors and more complex communication errors can jeopardise the development and execution of a swarm application. Confidentiality errors, derived from information leaks of confidential information, caused by mistakes in the specification of normal interactions between swarm participants are possible and frequent.

2.3.18.1 Tool Architecture

We propose an integrated approach that combines the editing, compiling and deploying of a swarm application with the verification of data confidentiality using information flow control (IFC) techniques. The verification of IFC[27] allows to deem unsafe some choreography designs that place/show information outside of the desired security compartments. Security compartments are traditionally defined using a lattice of security levels (mapped from the credentials given to principals upon authentication). Security levels go from secret (no one can see this information) to public (every principal can see this information) with several levels in between in an ordered way. In this work, we use value-dependent security levels[28] that allow for more flexibility and pragmatic implementation.

The approaches to enforcing IFC in DCR choreographies can be static, dynamic, or hybrid. Static verification works integrated in the compiler tool (T-WP3-03) and deems safe, or unsafe any annotated choreography. However, static verification is conservative and therefore gets too restrictive rapidly with the growing complexity of choreographies. The dynamic verification is more flexible, but needs exhaustive testing to build up confidence on the safety of the tested choreographies. Current research developments are walking towards hybrid approaches where dynamic verification is used when no certain unsafety is determined. Unsafe choreographies are rejected, safe choreographies are accepted without dynamic verification and grey cases are checked at runtime.

Our compiler translates a DCR choreography into a Babel protocol (a layer in a communication stack) where the local behaviour is specified, which can be executed in multiple network nodes by different participants in a swarm. Babel will also incorporate communication primitives from closely related tools like IFChannel (T-WP4-09) which implement verified protocols, e.g., by PSPSP (T-WP4-10).

The static verification of IFC is incorporated in the static validation phase (c.f. type checking) of the DCR choreography editor (T-WP3-03). Another check that is performed at this stage is the projectability check, which is the base for the projection process, producing separate instances of Java code to run local behaviour of each participant in the swarm.

An interesting, and convenient, result of this work is that projectability preserves the basic IFC property (non-interference). Future developments of this tool will implement dynamic checks whenever static checks are not 100% free of false positives. The tool will generate

[27] Declarative Choreographies with Data, Time and Information Flow Security, Eduardo Geraldo, João Costa Seco, Thomas Troels Hildebrandt. Under submission. 2024.
[28] Eduardo Geraldo, João Costa Seco, Thomas T. Hildebrandt: Data-Dependent Confidentiality in DCR Graphs. PPDP 2023: 7:1-7:13

code that annotates data and events with security levels and checks that the transmitted data and visible events always fall into allowed security compartments.

2.3.18.2 Tool Operational Details

The tool, given the specification of a DCR choreography, produces Java code that integrates with the Babel (T-WP6-04) framework and implements the local behaviour of each of the participants. The verification of information flow control happens during the type checking phase and may yield errors that are potentially linked to the breaking of data confidentiality.

2.4 WP5: Decentralised Machine Learning

An important aspect of the development of a swarm application within the TaRDIS toolbox is to provide the developer the possibility of utilising federated machine learning training strategies. On top of that, TaRDIS also offers an intelligent swarm orchestrator, that is capable of optimising network orchestration by means of federated learning, and machine learning inference agents, meant for deriving valuable information from target data sets.

The federated ML approaches within the TaRDIS architecture enable training on data sets that are distributed in diverse locations. The developer needs to specify the target data set and choose an approach for training. This corresponds to the requirements RF-WP5-FLALG-01 (this requirement details the FL framework and FL algorithm that will implement the decentralised training process) and RF-WP5-FLALG-05 (this requirement describes the diversity of the ML algorithmic groups that can be implemented through the Tardis toolkit, e.g., supervised learning algorithms), from D2.2. Specific requirements and Tardis toolkit functionalities can be also exploited in case of training a Reinforcement Learning (RL) agent. In this case, the developer needs to train the RL model in a simulated or digital twin environment (RF-WP5-RLALG-01), before deploying the agent in the real system. Additionally, the system offers data preparation functionalities, where it is possible to select and apply some preprocessing tasks that can be applied in a federated way. This is directly connected to the RF-WP5-FLALG-03 requirement from D2.2. After the process of preprocessing, the developer might use the prepared data for model training, by selecting the task to be solved and choosing a model and algorithm. The system guides the developer through the process of setting up the training, and offers a default, but configurable set of hyper parameter values. It should be noted that the orchestration of the resources is also provided by using inherent ML methods based on reinforcement learning, for instance task offloading schemes inside the swarm (RF-WP5-RLALG-02, RF-WP5-RLALG-03). The trained model can then serve as a base for the inference functionality, in order to gain some valuable conclusions on a target data set (this addresses the requirement RF-WP5-FLALG-04, from D2.2). Furthermore, the developer can leverage the functionality of the TaRDIS toolkit that provides a lightweight version of the ML model in terms of complexity and computational resources required for inference (RF-WP5-FL-ALG-06). We illustrate the usage of these functionality within TaRDIS use cases with the Actyx use case, specifically with detection of anomalies in the workflows in a smart factory environment. The process starts with applying pseudo-labeling on the data set of the workflows at the time of a new workflow acquisition, as a preprocessing step. The pseudo-labelling is done automatically based on predefined domain specific rules but is realised by the FL preprocessing TaRDIS module. This serves as a base for performing anomaly detection over the workflows. Subsequently, an anomaly detection model is trained based on the collected pseudo-labelled workflow historical data using one of the FL training modules. An incremental model retraining possibility is also included and supported by the FL training modules (see the requirement RF-WP5-FLALG-02). Next, in real time, the FL inference module performs inference (anomaly detection) for the newly acquired workflow data point. All the mentioned concepts correspond to the generic use case requirement RF-WP5-GEN-01 from D2.2. The

details regarding the connection between WP5 tools and requirements are shown in the tables below.

2.4.1 Framework supporting AI/ML programming primitives

Flower is an open-source framework that enables large-scale FL implementations and is language and ML-framework agnostic. It is highly extensible and widely used in research and production deployment. Within Task 5.1, FL implementations in Flower were completed with three different federation algorithms (applied by the aggregator): FedAvg, Personalized FL with Moreau envelopes (pFedMe) that was recently proposed in the literature, and a variation of the latter method called pFedMeNew (developed as part of the efforts of Task 5.1). The federation algorithms pFedMe and pFedMeNew differ in local iterations and batch sampling. Both algorithms perform a defined number of global rounds, and a defined number of local rounds. Inside the local rounds, the algorithms run K inner iterations. The proposed algorithm pFedMeNew takes a new batch inside every inner iteration, while pFedMe, takes a batch and performs the inner iterations on it. These algorithm implementations are examples of the Flower-based FL model training tool algorithms, within Task 5.1. The other framework of interest regarding Task 5.1 is PTB-FLA that is also meant for the development of FL algorithms. It can be considered as a development paradigm that serves as a FLA developer guide through the process of developing a target FLA using PTB-FLA. Finally, a Decentralised Federated Learning Framework (Fedra) tool was developed to provide federated learning based on a decentralised, anonymous peer-to-peer communication framework, while securing model weight exchange, upholding privacy and data ownership. To summarise, the following tools are presented under Task 5.1: Flower-based FL model training, Data preparation for Flower-based FL model training, Flower-based FL model inference and evaluation, PTB-FLA-based FL model training and Fedra.

T-WP5-01: Flower-based FL model training
T-WP5-04: PTB-FLA-based FL model training
T-WP5-09: Decentralised Federated Learning Framework (Fedra)

FL model training requirements

Requirement ID and name	T-WP5-01	T-WP5-04	T-WP5-09
RF-WP5-FL-ALG-01, RF-WP5-GEN-01 a list of provided FL algorithms	✔	✔	✔
RF-WP5-FL-ALG-02 support for incremental model retraining within FL algorithms	✔	✔	
RF-WP5-FL-ALG-05 support diverse ML algorithms in decentralised frameworks	✔	✔	✔
RF-GMV-02 Interaction between satellites in the distributed model	✔	✔	✔
RF-GMV-11 Implementation of an FL model for distributed ODTS	✔	✔	✔

Requirement ID and name	T-WP5-01	T-WP5-04	T-WP5-09
RF-GMV-12 Substitution of the orbit propagator with a ML/FL model	✔	✔	✔

The following tools offer support for data preparation in the context of federated machine learning within an intelligent heterogeneous swarm.

T-WP5-02: Data preparation for Flower-based FL model training

FL training data preparation requirements

Requirement ID and name	T-WP5-02
RF-WP5-FL-ALG-03 data preprocessing facility for FL	✔

2.4.2 AI-driven planning, deployment & orchestration framework

A simulation for training a Reinforcement Learning (RL) agent has been built within Task 5.2. It is divided into discrete time steps and can handle various network configurations, i.e., peer-to-peer networks but also networks where there is a hierarchy among the nodes. The simulator runs in Java and must be wrapped into a Python environment following the interface defined as a Markov Decision Process or a Markov Game, in the case of decentralised decision-making. Such RL training environments are embedded in an RL training framework, in our case, we chose PettingZoo. The developed environment models task offloading-based orchestration. It is an open platform that the team plans to extend to broader action spaces. Since the simulation alone is insufficient for training a Reinforcement Learning (RL) agent, an environment following the Markov Decision Process (MDP) framework is created as well. This environment was created using the PettingZoo framework in Python. The purpose of this developed tool is to learn to orchestrate the network in a federated way. Within Task 5.2, the tool Federated AI Network Orchestrator (FAUNO) is presented.

T-WP5-05: Federated AI Network Orchestrator (FAUNO)

FL intelligent orchestration requirements

requirement ID and name	T-WP5-05
RF-WP5-RLALG-01 A simulation environment for training of RL agents	✔
RF-WP5-RLALG-02 Centralized RL agent for task offloading	✔
RF-WP5-RLALG-03 Decentralized RL agent for task offloading	✔
RF-GMV-08 Optimization of the ISL connectivity scheme	✔

2.4.3 Library of lightweight and energy efficient ML techniques

Within Task 5.3, different techniques for making a ML model more lightweight and energy-efficient are introduced. These techniques are related to the use of Deep Neural Networks (DNNs), as the DNNs require the most computational resources for operation at swarm systems or at swarm devices. The three methods of interest are: Early-Exit of inference techniques (T-WP5-06), Knowledge Distillation (T-WP5-07) and Pruning (T-WP5-08). Specifically, the Early-Exit (T-WP5-06) tool provides a segmentation of the DNN architecture in the system topology, taking into consideration the architecture of the IoT-edge-cloud continuum. Its functionality relies on including earlier exits in the DNN architecture (apart from the final output layer) that will enable latency-aware inference of the model. Moreover, the Knowledge Distillation (T-WP5-07) tool is based on providing a more lightweight "student" model version of the original DNN by compressing the training experience of the latter "teacher" model. Furthermore, the Pruning (T-WP5-08) tool aims to also provide a more lightweight version of the original DNN by removing neuron weights and connections that have negligible impact on the performance of the network, hence optimising its computational energy efficiency.

T-WP5-03: Flower-based FL model inference and evaluation
T-WP5-06: Early-Exit (Lightweight Functionality)
T-WP5-07: Knowledge Distillation (Lightweight Functionality)
T-WP5-08: Pruning (Lightweight Functionality)

FL inference requirements

Requirement ID and name	**T-WP5-03**	**T-WP5-06**	**T-WP5-07**	**T-WP5-08**
RF-WP5-FL-ALG-04 support for ML inference and evaluation	✔			
RF-WP5-FL-ALG-06 lightweight techniques for ML training and inference		✔	✔	✔
RNF-TID-04 Energy-efficient training/inference		✔	✔	✔
RNF-TID-02 Federated learning with low impact on user experience		✔	✔	✔
RF-GMV-09 ODTS algorithm/ML model efficiency		✔	✔	✔
RF-GMV-13 Feasibility of the ODTS ML model for on-board implementation		✔	✔	✔
RF-GMV-14 Feasibility of the orbit propagation ML/FL model for on-board implementation		✔	✔	✔

Funded by the European Union

Requirement ID and name	T-WP5-03	T-WP5-06	T-WP5-07	T-WP5-08
RF-GMV-15 Feasibility of the on-board implementation of the RL agents for ISL scheduling		✔	✔	✔

2.4.4 T-WP5-01 Flower-based FL model training

This tool provides federated machine learning (FL) solutions and enables model training. It provides a simple usage of FL solutions, without requiring expert knowledge. The developer needs to specify only the task that should be solved and the mandatory minimal input parameters. Based on the task, the system offers a list of available ML models and algorithms. For instance, if the user selects supervised classification from the list of tasks, the tool shows the available ML models for the task, as Logistic regression, KNN, SVM, etc., and the available algorithms, as FedAvg, pFedMe, etc. The user can pick a model and algorithm on their own, or they can use the default, preselected options. Regarding the input parameters, the developers need to specify the training data set, while the hyperparameters and advanced setup parameters can be used with their default values, or they can be adjusted according to the needs. When the developer finishes the specification and adjustment of the input parameters, the tool performs an analysis that ensures that the required setup is applicable. This ensures a highly customizable and also reliable approach, that is easy to use while supporting the developers' decisions and that can be easily utilised, by following the proposed solutions.

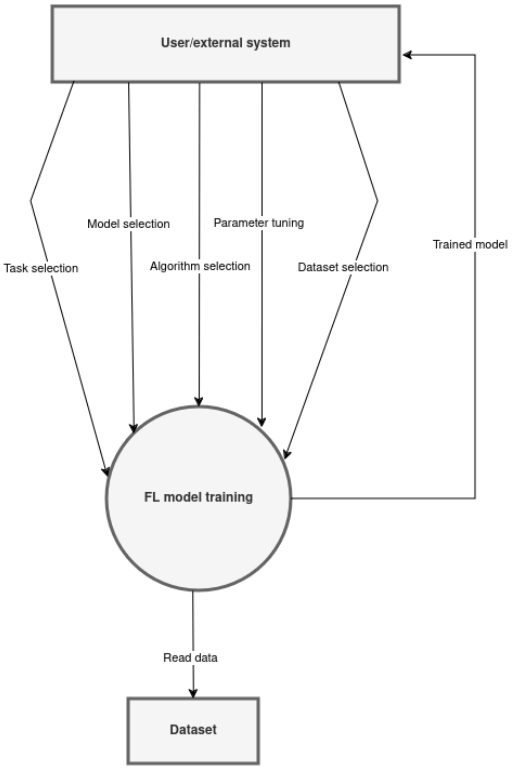
The implementations of customised strategies for FL algorithms are achieved by utilising the features that the Flower framework offers. It provides model training for the selected approach. This process of initiating the model training is simplified by letting the user choose the task that needs to be solved (e.g., supervised binary classification, supervised M-ary classification, regression…). For the selected task, the tool can offer a list of applicable models (e.g., neural network, logistic regression…), and a list of available algorithms (FedAvg, pFedMe…) as well. To summarise, regarding the inputs for this tool, the target FL algorithm needs to be selected first. This can be done incrementally, by selecting a task, then a suitable model and algorithm for the task. Then, the target data set needs to be determined. Additionally, the hyperparameters' values can be adjusted and some advanced setup options can be also set (e.g., the number of clients). The output is the trained ML model and the status of the finished training.

The tool is designed to provide an AI/ML library with a variety of decentralised solutions. However, the library is also meant to include some use case specific solutions for TaRDIS, such as anomaly detection of workflows with noisy labels for the ACT use case. This tool is tightly coupled with two other TaRDIS tools: The data preparation tool for Flower-based FL model training (T-WP5-02) and the Flower-based FL model inference and evaluation (T-WP5-03). The tool T-WP5-02 is meant to provide data format that is suitable for FL model training, while the tool T-WP5-03 offers solutions for gaining conclusions, by means of inference and evaluation.

2.4.4.1 Tool Architecture

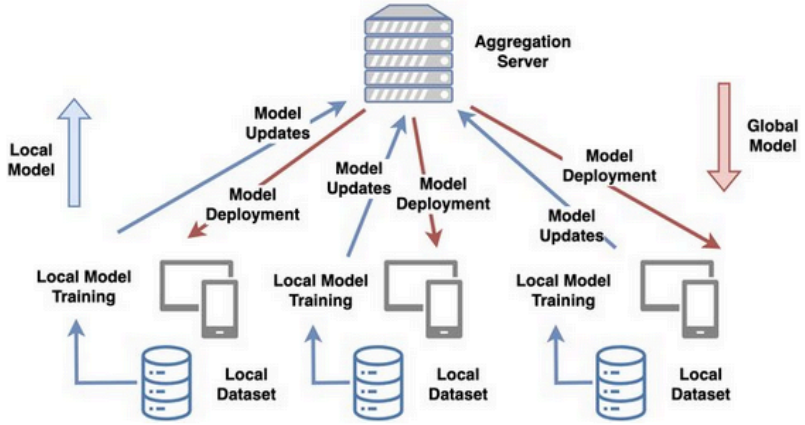
The context diagram of the tool is represented below. The tool can be used by a user or another tool that is responsible for initiating the training process. This means selecting the task and the corresponding model and algorithm, as well as adjusting the parameters (optionally) and picking a data set. The tool is then reading the target data set and performing

the training process, which results with a trained model that is being saved in a file on a predefined location.



Flower-based FL training architecture

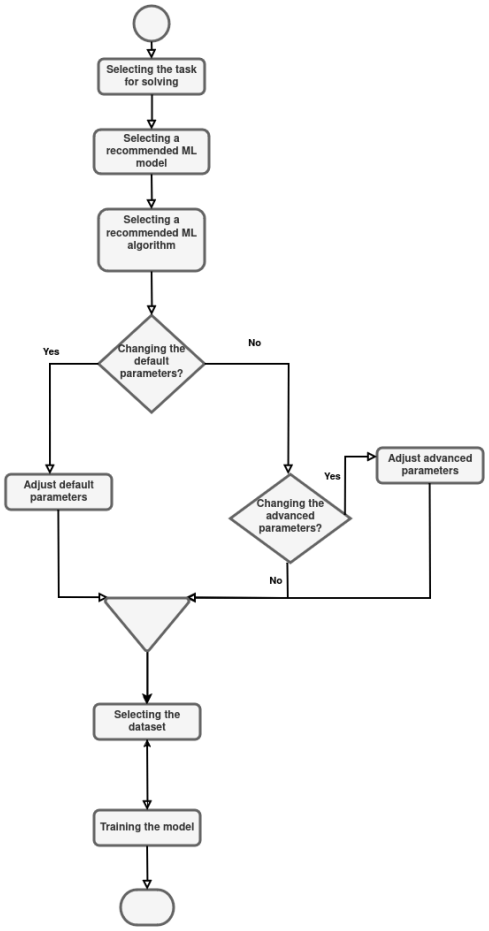
The tool does not require specific deployment by its structure. However, since it represents a library of FL solutions, a classic deployment view of federated learning process as an example shown below[29], corresponds to this tool, and hence the algorithms within it, as well.



[29] Zhang, Hongyi & Bosch, Jan & Olsson, Helena. (2023). EdgeFL: A Lightweight Decentralized Federated Learning Framework.

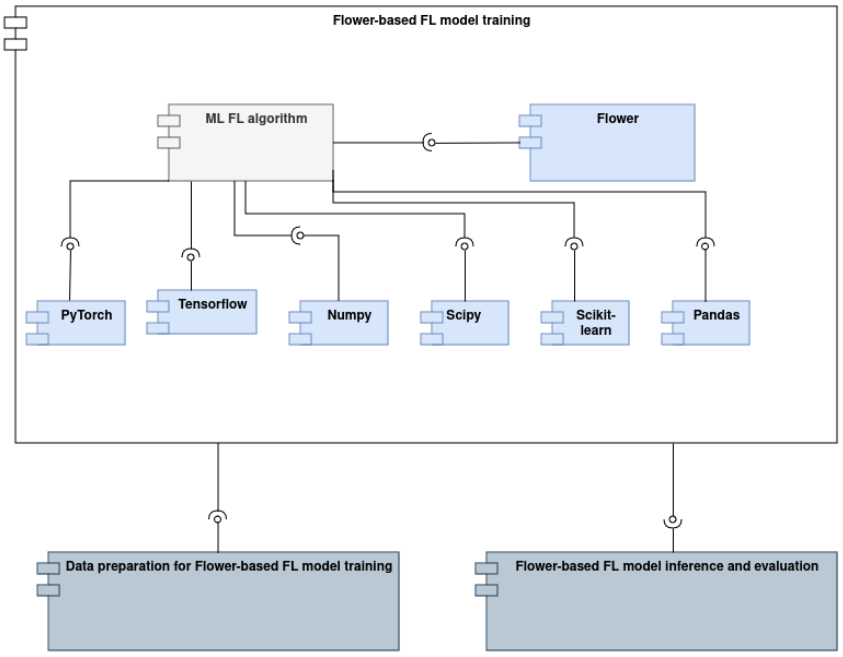
Flower-based FL training data flow

The tool has a compact design, it represents a unique module, that relies on some available components as Flower and specific Python libraries. The diagram below represents the workflow of the usage of this tool, from the aspect of an end user. The user needs to select the task and the ML model and algorithm. Then, the hyperparameters may be used with default values or can be adjusted according to the needs. Similarly, for the advanced parameters, such as number of clients, either default values could be used, or the user could specify custom values. Finally, after specifying the data set, the training process can start, which will result with a trained model. The training process can be initiated by another tool as well, by issuing a command with the needed arguments.



Flower-based FL training workflow

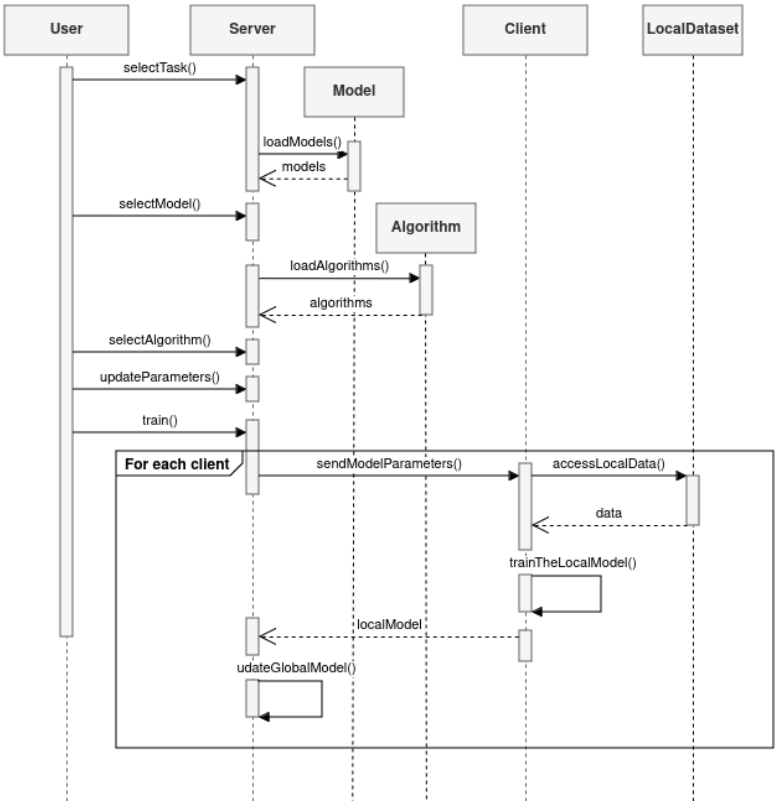
The diagram below shows the composition of the tool, consisting of the specific algorithmic implementations, connected with the Flower framework and common Python libraries. The tool uses the Data preparation tool and produces a model that can be an input for the inference and evaluation tool.



Flower-based FL training tool composition

2.4.4.2 Tool Dynamic Behaviour

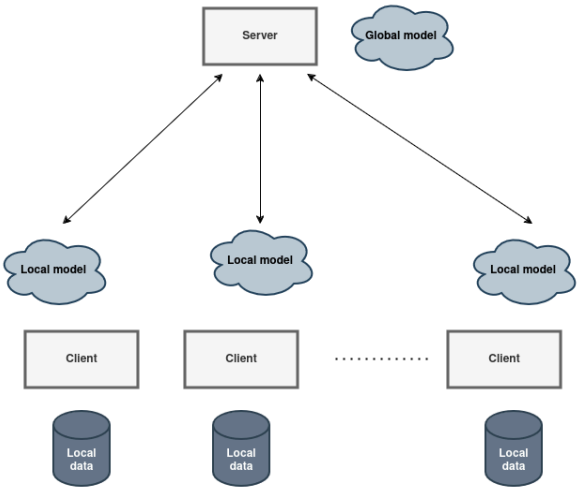
A diagram displaying the sequence of communication during the tool usage and training process is shown below. The user initiates the process by selecting a task. Then, the server offers the corresponding model list. After choosing a model, the server shows the list of available algorithms. The user picks an algorithm, updates the parameters and confirms the start of training. The server then sends the model parameters to each client. The clients train the model on their own data and provide their local models to the server, in order to update the global model.



Flower-based FL training dynamic behaviour

2.4.4.3 Tool Persistence

The diagram below shows the typical federated learning setup, where each client operates on its local data, in order to obtain a local model, while the server updates the global model, based on the responses from the clients.



Flower-based FL training persistence

The tool does not require a database, there are no persistent entities related to the tool. The trained model is being saved in a predefined location in a file, when the training ends.

2.4.4.4 Tool Operational details

As already stated, the tool is relying on the Flower framework and on some well-known Python libraries: Flower, PyTorch (torch and torchvision), Tensorflow, Numpy, Scipy, Scikit-learn, and Pandas. It can be possibly used through a GUI, or by default through a command line interface (see the details in D3.2). It sets the foundations for model training by interacting with the developer and guiding them through the process. The tool is constantly evolving, as new algorithms are being implemented and offered as possible ML solutions.

2.4.5 T-WP5-02 Data preparation for Flower-based FL model training

This tool enables data preprocessing and preparation for ML training. The data set that the developer wants to use for ML model training usually does not come in a clean form that is appropriate for model training. Usually, it contains a variety of irregularities, for instance duplicates, missing values, outliers, etc. This tool provides a convenient way to solve these common issues. Besides these irregularities, some additional, custom data preparation approaches may be needed. An example for this is the case when the model requires labelled data, but the data set does not contain labels. In this case, pseudo-labeling approaches could provide a solution. This tool is straightforward to use for the developers. It only requires specifying the data set and the preprocessing task to be performed, with the possibility for customization where applicable.

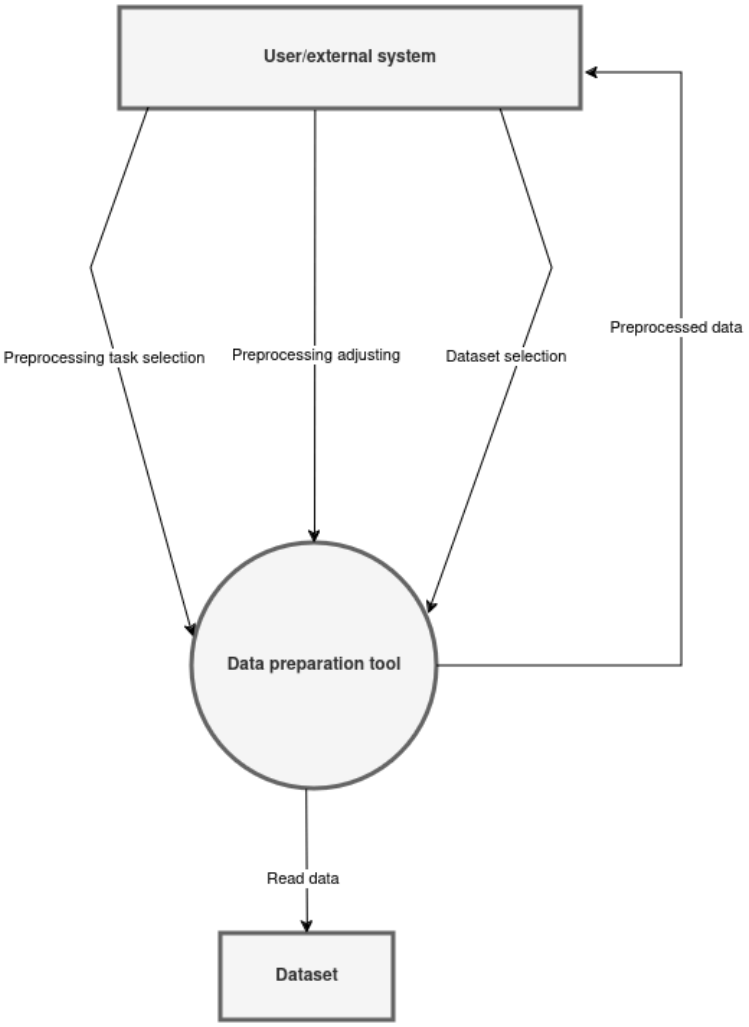
The tool is envisioned to ease the process of FL model training, by partially automating (meaning that the user still needs to specify the approach and tasks to be performed) the input data preparation, which is an inevitable step towards successful ML model training. The process of preparing data for ML training consists of applying different preprocessing tasks that eliminate common irregularities (e.g., missing values, outliers, duplicates…). It is also possible that the training algorithm requires additional features, which can be also handled during preprocessing. This tool offers the possibility to transform such a raw data set into a form that can be used by FL model training algorithms. It is tightly coupled with the tool: The Flower-based FL model training (T-WP5-01), as it is meant to provide data format that is suitable for T-WP5-01, so it can be generally used as a predecessor to FL model training. However, it also finds its usage in the ACT TaRDIS use case, where the need for generating pseudo labels emerges (see D5.1).

The required inputs for this tool are the selected preprocessing approach/es, the data set and the preprocessing tasks (e.g., cleansing, profiling, transformation…) with possible adjustments. The output is the prepared data set, with some additional information about the status and the tasks that were carried out.

2.4.5.1 Tool Architecture

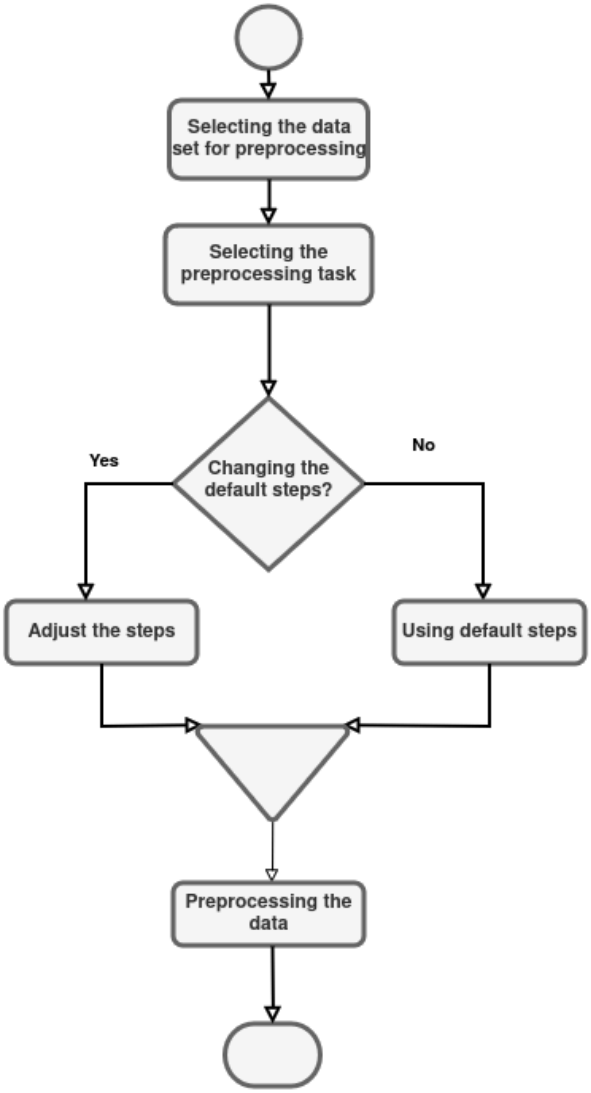
The context diagram of the tool is represented below. The tool can be used by a user or another tool that is responsible for initiating the preprocessing. This means selecting the preprocessing tasks with adjustments (optionally) and picking a data set. The tool is then reading the target data set and performing the preprocessing and preparation, which result with a transformed data set that is being saved in a file on a predefined location.

Funded by the European Union



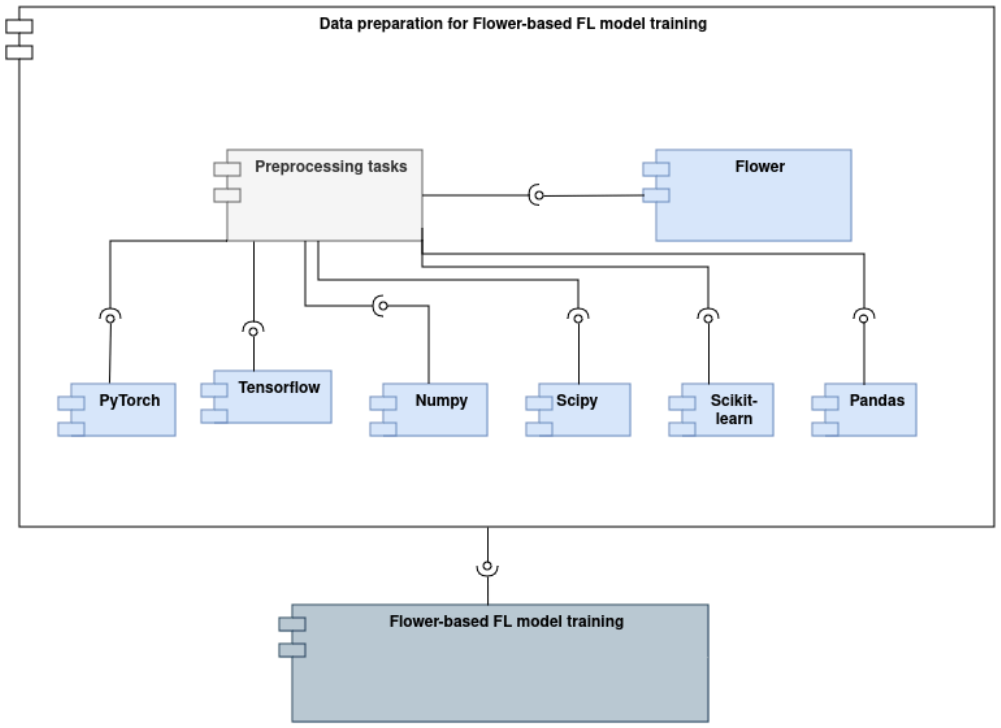
Flower-based FL data preparation architecture

The tool has a compact design, it represents a unique module, that relies on some available components as Flower and specific Python libraries. The diagram below displays the information flow of initialising and starting the preprocessing. The user needs to specify the data set, the preprocessing task, and optionally the adjustment of the steps to be performed. After that, the preprocessing can start, which results with a transformed data set. In a federated setting, this means that each client can preprocess its local data, based on the defined setup. The process can be initiated by another tool as well, by issuing a command with the needed arguments.



Flower-based FL data preparation workflow

The diagram below shows the composition of the tool, consisting of the specific preprocessing tasks implementations, connected with some needed external libraries. The tool produces a transformed data set that can be used by the FL model training tool.



Flower-based FL data preparation components

2.4.5.2 Tool Dynamic behaviour

The sequence of steps performed during setting up the preprocessing is simply consisting of choosing and adjusting the task, choosing the data set and starting the performing of the preprocessing. Each client can apply the specified tasks on its local, in a federated setting.

2.4.5.3 Tool Persistence

The data set used for FL is not a compact one. Instead, each client has its own local data, where the preprocessing happens. The data distribution corresponds to the classic FL setup. The preprocessed data is being saved in a predefined location in a file, when the preprocessing ends and it can be used further for the needed ML model training.

2.4.5.4 Tool Operational details

The tool is relying on the same external libraries as described for the tool T-WP5-01, i.e., on the Flower framework and on some well-known Python libraries: Flower, PyTorch (torch and torchvision), Tensorflow, Numpy, Scipy, Scikit-learn, and Pandas. The tool can be possibly used through a GUI, or by default through a command line interface. (see the details in D3.2). The tool is constantly evolving, as new preprocessing tasks are being implemented and added to the list of offered tasks.

2.4.6 T-WP5-03 Flower-based FL model inference and evaluation

This tool enables gaining output for the relevant data on a trained model. The tool provides a convenient way for the developer to gain insights into the quality of the trained model and to obtain valuable conclusions. The developer can use this tool after obtaining the model, by simply selecting the trained model, the test data set and the type of inference or evaluation. The tool provides the needed outputs, such as gaining predictions, forecasting, performing

Funded by the European Union

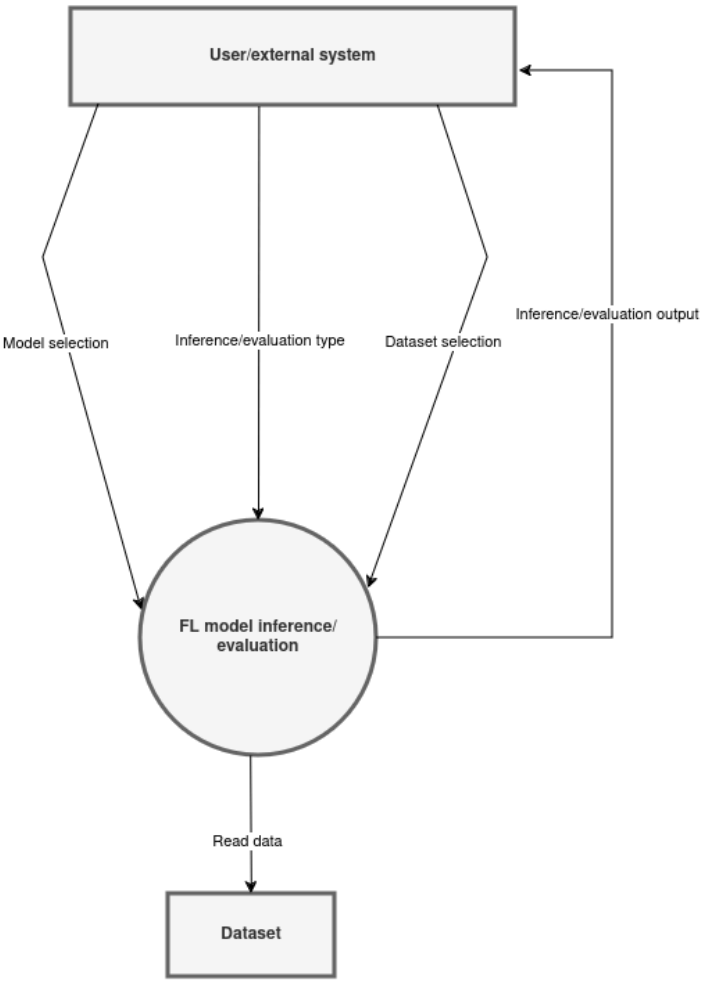
anomaly detection, application of different metrics, etc., by simply selecting the appropriate technique.

The tool is envisioned to provide some important conclusions on a data set, regarding a trained FL model. This represents an important step in FL, as it provides the needed insights. The tool achieves this by means of inference and different kinds of evaluations.

Inference and evaluation can be generally used for the obtained trained FL models.The tool also finds its usage within the TaRDIS ACT use case for instance. It is tightly coupled with the tool: The Flower-based FL model training (T-WP5-01), as it is meant to provide valuable outputs, based on the model trained within T-WP5-01. The tool requires the following inputs: the trained model, the input parameters, the target data and the type of inference or evaluation that is of interest to be done. The output depends on the type of inference/evaluation chosen. It can be a set of predictions, metrics, accuracies, etc.

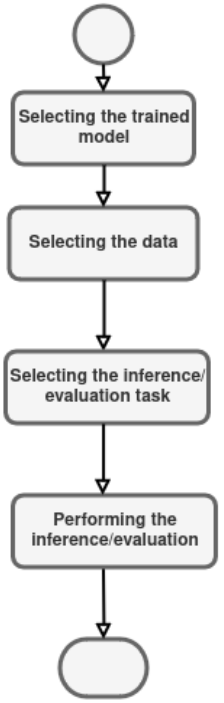
2.4.6.1 Tool Architecture

The context diagram of the tool is shown below. The user needs to select a trained model, a target dataset and the type of inference/evaluation. Depending on the selection, the tool provides an appropriate output.



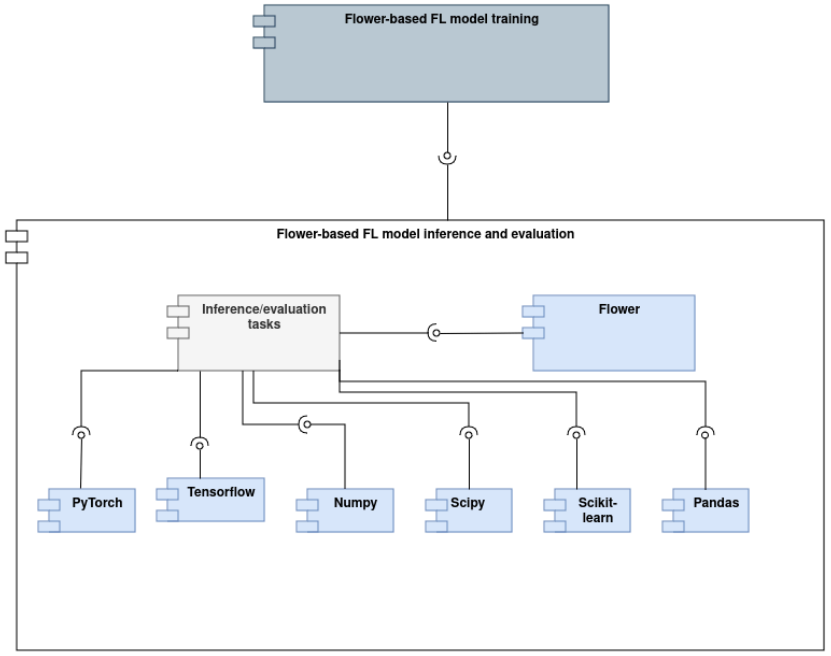
Flower-based FL inference architecture

The tool relies on some available components such as Flower and specific Python libraries. It can perform inference in a federated setting, by applying the user specified inputs to the client nodes. The workflow within this tool is straightforward and is shown below. It only requires specifying the model, the data and the inference/evaluation type, in order to start the process. The process can be initiated by a user or by another tool as well, by issuing a command with the needed arguments.



Flower-based FL inference workflow

The diagram below shows the composition of the tool, consisting of the specific inference/evaluation tasks implementations, connected with some needed external libraries. The tool uses the model gained from the FL training process and produces the expected output, that is highly dependent on the type of inference or evaluation.



Flower-based FL inference components

2.4.6.2 Tool Dynamic behaviour

The tool has only 2 potential states: receiving setup info and performing inference/evaluation. The sequence of steps performed during the inference/evaluation is simply consisting of choosing the task, choosing the data set and model, and performing the inference/evaluation.

2.4.6.3 Tool Persistence

In the cases when the data for inference is distributed locally on clients, the data distribution corresponds to the usual federated setup. Each client can run the inference tasks on its local data, or it can be done in a centralised manner, when applicable. The output is being saved in a predefined location in files, when the preprocessing ends.

2.4.6.4 Tool Operational details

The tool is relying on the same external libraries as described for T-WP5-01, i.e., on the Flower framework and on some well-known Python libraries: Flower, PyTorch (torch and torchvision), Tensorflow, Numpy, Scipy, Scikit-learn, and Pandas. It can be possibly used through a GUI, or by default through a command line interface (see the details in D3.2). The tool is constantly evolving, as new inference/evaluation tasks are being implemented.

2.4.7 T-WP5-04 PTB-FLA-based FL model training

Python Testbed for Federated Learning Algorithms (PTB-FLA)[30] is a tool developed with the primary intention to be used as a framework for developing federated learning algorithms (FLAs), or more precisely as a runtime (or execution) environment for FLAs under development on a single computer (i.e., localhost). PTB-FLA supports both centralised and decentralised federated learning algorithms.

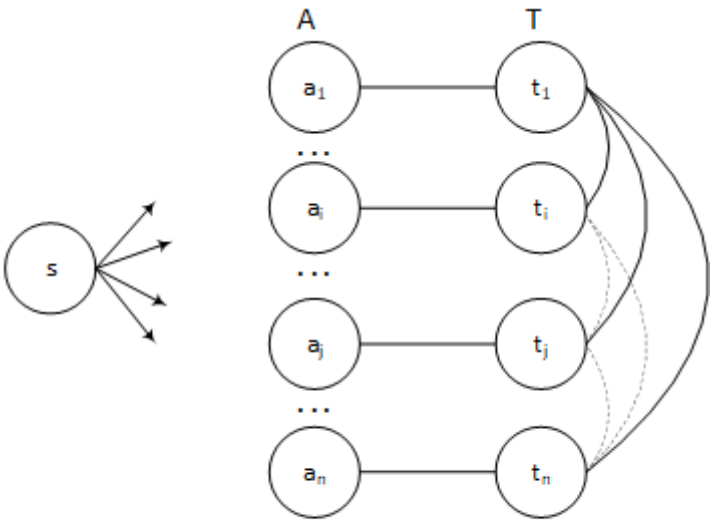
[30] M. Popovic, M. Popovic, I. Kastelan, M. Djukic and S. Ghilezan, "A Simple Python Testbed for Federated Learning Algorithms," *2023 Zooming Innovation in Consumer Technologies Conference (ZINC)*, Novi Sad, Serbia, 2023, pp. 148-153, doi: 10.1109/ZINC58345.2023.10173859

PTB-FLA is written in pure Python, which means that it only depends on the standard Python packages, such as the package multiprocessing, and it was intentionally written this way for the following two reasons: (i) to keep the application footprint small so to fit to IoTs, and (ii) to keep installation as simple as possible (with no external dependencies).

PTB-FLA enforces two restrictions that must be obeyed by the algorithm developers. First, a developer writes a single application program, which is later instantiated and launched by the PTB-FLA launcher as a set of independent processes whose behaviour depends on the process id. Second, a developer only writes callback functions for the client and the server roles, which are then called by the generic federated learning algorithms hidden inside PTB-FLA.

2.4.7.1 Tool Architecture

The block diagram of the PTB-FLA system architecture is shown below:



Legend: s – application launcher, A – application, ai – application program instance, T – testbed, ti – testbed program instance.

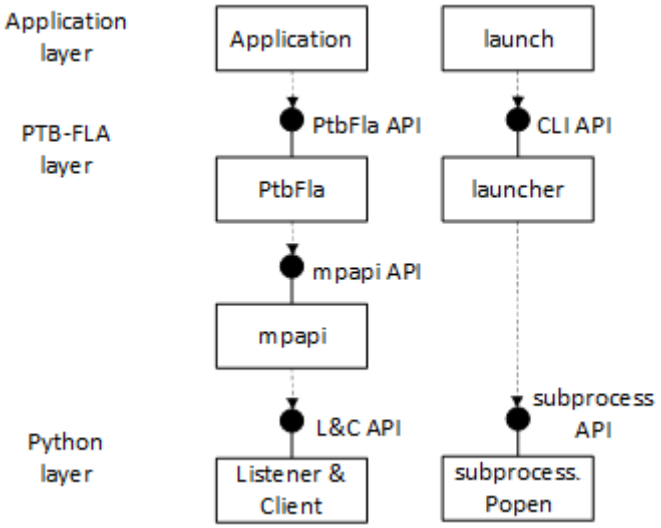
PTB-FLA training block diagram

The PTB-FLA system architecture, consists of the application launcher process *s,* the distributed application $A = \{a_1, a_2, \ldots, a_n\}$, which is a set of application program instances a_i, and the distributed testbed $T = \{t_1, t_2, \ldots, t_n\}$, which is a set of testbed instances t_i, where $i = 1, 2, \ldots, n$, and n is the number of instances in both A and T.

The system starts as follows. Once the launcher process s is manually started from the command line interface, it instantiates n application program instances a_i, $i = 1, 2, \ldots, n$, and launches them as n independent processes. Each application program instance a_i in turn creates its testbed instance t_i. At the end, the testbed instances conduct the startup handshake by exchanging hello messages.

During normal system operation, the distributed application A uses the distributed testbed T to execute the distributed algorithm, which is specified by the callback functions within the application program (i.e., in the application Python modules).

The UML class diagram of the PTB-FLA system architecture is as shown below:



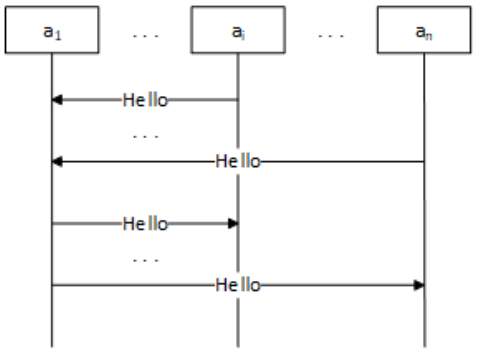
PTB-FLA training architecture

The PTB-FLA system architecture comprises three layers: the distributed application layer on top (comprising the application modules and the console script launch), the PTB-FLA layer (comprising the class *PtbFla* in the module *ptbfla* and the modules *mpapi* and *launcher*), and the Python layer (including classes *Process*, *Queue*, and *Listener* & *Client* from the package *multiprocessing* and *Popen* from the package *subprocess*).

The console script *launch* uses the module *launcher* (which in turn uses *Popen*) to launch the distributed application comprising n independent processes, p_i, i = 1, 2, …, n, where each p_i comprises the corresponding pair of instances (a_i, t_i) and executes in a separate terminal (i.e., window). On the other hand, the application module uses the PtbFla API (comprising PtbFla functions) to create or destroy a testbed instance (by calling the constructor or the destructor) and to conduct its role in the distributed algorithm execution (by calling the API function *fl_centralized* or the API function *fl_decentralized*). The API functions *fl_centralized* and *fl_decentralized*, within an instance t_i, use the module *mpapi* (*mpapi* is the abbreviation of the term *message passing API*) to communicate with other instances. The module *mpapi* in turn instantiates the Python multiprocessing classes *Listener* and *Client* to create the *mpapi* server and the *mpapi* client, which are hidden with the module *mpapi* and should not be confused with the server and client roles in the federated learning algorithms.

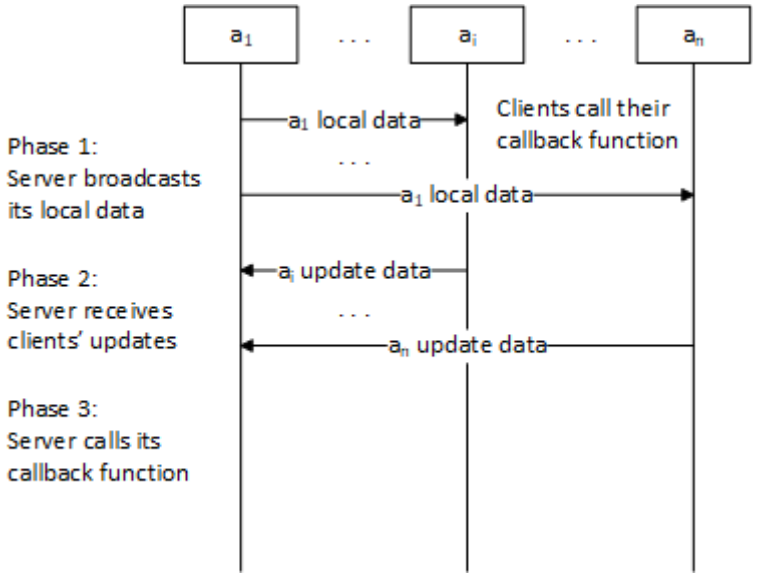
2.4.7.2 Tool Dynamic Behaviour

The system startup handshake has two phases. In the first phase, the instance a_1 is waiting to receive (n – 1) Hello messages from all other instances a_i, i = 2, …, n, and in the second phase, the instance a_1 broadcasts the message Hello to all other instances.



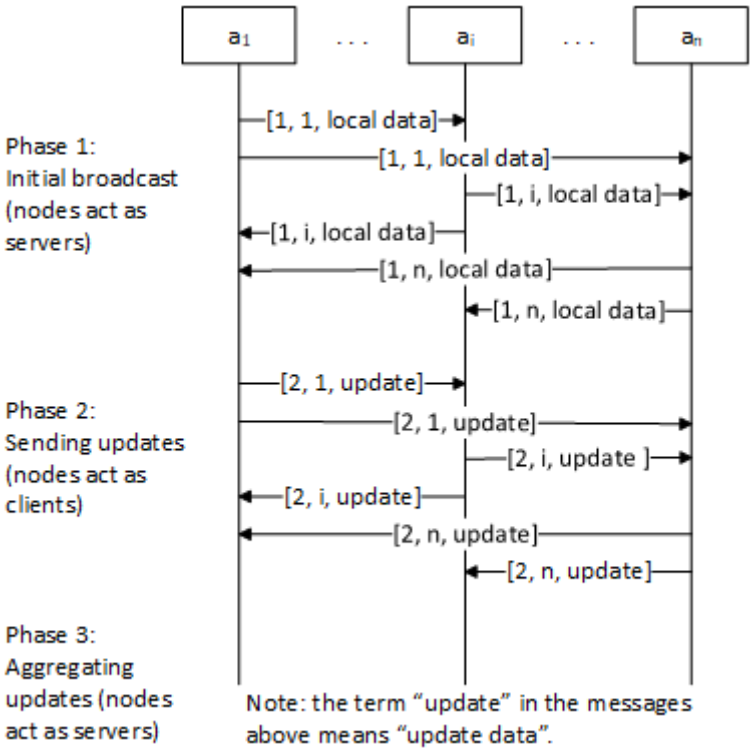
PTB-FLA training communication phase 1

The generic centralised one-shot FLA has three phases. Let's assume that the instance a_1 is the server and the other instances a_i, i = 2, …, n, are the clients. In the first phase, the server broadcasts its local data to the clients, which in turn call their callback function to get the update data and store the update data locally. In the second phase, the server receives the update data from all the clients, and in the third phase, the server calls its callback function to get its update data (e.g., aggregated data) and stores it locally. Finally, all the instances return their new local data as their results.



PTB-FLA training communication phase 2

The generic decentralised one-shot FLA has three phases. In the first phase, each instance acts as a server, and it sends its local data to all its neighbours. These messages have the sequence number 1, and each instance sends (n – 1) such messages. Note that each instance is also the destination for (n – 1) such messages. In the second phase, each instance acts as a client, and it may receive either of the messages with the sequence numbers 1 (sent in the first phase) or 2 (sent during the second phase). If the instance receives a message from the second phase, it just stores it in a buffer for later processing, whereas if the instance receives a message from the first phase, it calls the client callback function to get the update data, and then sends the reply to the message source. In the reply, the instance sets the message fields as follows: the field sequence number to 2, the field message source address to its own address, and the field data to update data. Note that during the second phase, the instance does not update its local data, it just passes the update data it got from the client callback function.



PTB-FLA training communication phase 3

2.4.7.3 Tool Operational details

The tool required Python 3 and no other dependencies. The installation is easy - just install the PTB-FLA package using pip.

The PtbFla API comprises the following four functions (the variable after "/" is the function return value):

1. `PtbFla(noNodes, nodeId, flSrvId=0) / None`
2. `fl_centralized(sfun, cfun, ldata, pdata, noIters=1) / ret`
3. `fl_decentralized(sfun, cfun, ldata, pdata, noIters=1) / ret`
4. `PtbFla() / None`

The first is the constructor that is called as a global function and does not have a return value, the second and the third are member functions that are called on the instance of PtbFla, and the fourth is the destructor that is called implicitly by the garbage collector or explicitly when deleting an object.

The arguments are as follows: *noNodes* is the number of nodes (or processes), *nodeId* is the node identification, *flSrvId* is the server id (default is 0; this argument is used by the function fl_centralized), *sfun* is the server callback function, *cfun* is the client callback function, *ldata* is the initial local data, *pdata* is the private data, and *noIters* is the number of iterations that is by default equal to 1 (for the so called one-shot algorithms), i.e., if the calling function does

not specify it, it will be internally set to 1. The return value *ret* is the node final local data. Data (*ldata* and *pdata*) is application specific.

Typically, local data (*ldata*) is a machine learning model, whereas private data (*pdata*) is training data that is used to train the model. For example, in case of a simple linear regression i.e., straight-line fit to data, $y = ax + b$, the machine learning model is the pair of coefficients (a, b) where a is the slope and b is the intercept, whereas the training data is the given array of points i.e., pairs (x_i, y_i), $i = 1, \ldots, n$, where n is the number of points.

Normally, the testbed instances only exchange the local data (i.e., their local machine learning models) and they never send out the private data (that is how they guarantee the training data privacy). The private data is only passed to callback functions (within the same process instance) to immediately set them in their working context.

To use the tool, the developer needs to define the two callback functions and launch the application with the launcher. For example, to launch an example with 3 nodes in which the node 0 is the server node, the following command is needed:

```
launch src/examples/example1_fedd_mean.py 3 id 0
```

2.4.8 T-WP5-05 Federated AI Network Orchestrator (FAUNO)

The aim of this tool is to learn to orchestrate the network in a federated way. The Federated AI Network Orchestrator (FAUNO) is designed to optimise computer network orchestration through federated reinforcement learning (FRL). This tool leverages the principles of federated learning, where multiple nodes collaboratively train a model without sharing raw data, and reinforcement learning, which allows the system to learn optimal policies through trial and error interactions with the environment. By integrating these two approaches, FAUNO aims to enhance the efficiency and performance of distributed network systems.

The primary purpose of FAUNO is to provide a robust and scalable solution for managing and optimising network operations across multiple decentralised nodes. Traditional centralised approaches often face challenges related to latency, bandwidth constraints, and data privacy concerns. FAUNO addresses these issues by enabling nodes to learn and adapt locally while contributing to a global model that enhances overall network performance. This federated approach ensures data privacy and reduces the need for extensive data transfer, thus minimising latency and bandwidth usage.

In the context of TaRDIS use cases, FAUNO can be employed in various scenarios where network orchestration is crucial. These include dependencies/Interactions with other TaRDIS tools. FAUNO interacts with several other tools within the TaRDIS framework to achieve its objectives:

- **Data Aggregators**: These tools collect and preprocess data from various network nodes, providing the necessary inputs for the FRL algorithms.
- **Monitoring Systems**: Continuous feedback from network monitoring systems is crucial for the reinforcement learning component of FAUNO, enabling it to adapt and optimise in real-time.
- **Safety, correctness and analytical tools:** Post-implementation analysis and performance evaluation are conducted using analytical tools that provide insights into the effectiveness of FAUNO orchestration strategies. To be implemented.
- **Security and Integrity Modules:** Ensuring secure communication and data exchange between nodes is important, and FAUNO relies on integrated security modules to maintain data integrity and confidentiality. To be implemented.

The inputs for this tool can be listed as follows:

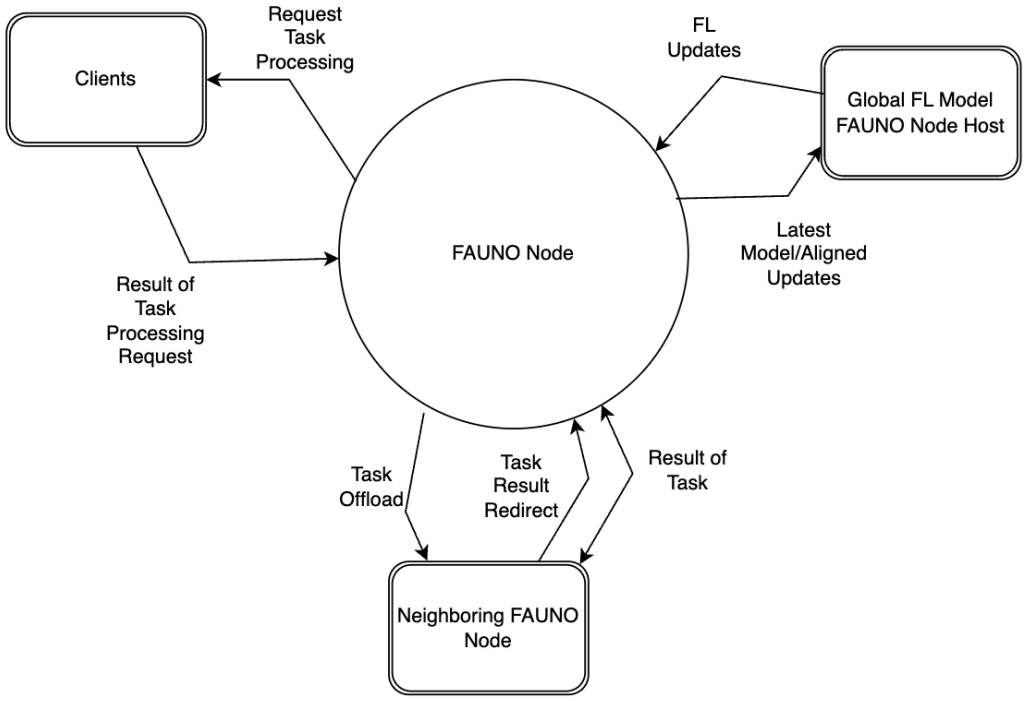
- **Local State Information**: Represents data about the current state of each node, including processing capabilities, queue sizes, and network conditions.
- **Task Specifications**: Includes details about the tasks to be processed or offloaded, such as computational requirements and deadlines.
- **Reward Signals**: Representing feedback from the environment indicating the success or failure of specific actions taken by the nodes.

The outputs for the tool are the following:

- **Optimised Policies:** The policies derived from the FRL process, which dictate how tasks should be allocated and processed across the network.
- **Performance Metrics:** Quantitative data on network performance improvements, such as reduced latency, increased throughput, and better resource utilization.

2.4.8.1 Tool Architecture

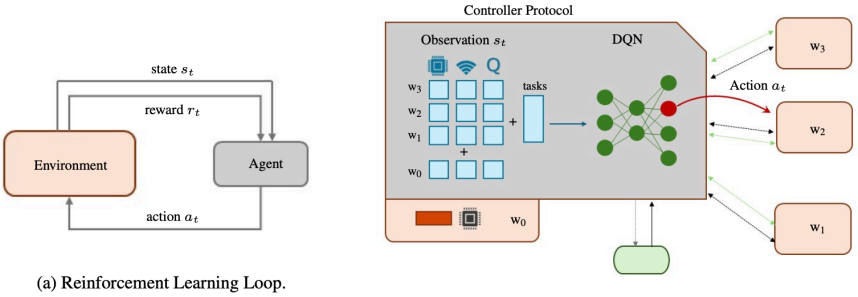
System Context diagram



FAUNO tool architecture

This diagram is showing the high-level interaction between FAUNO Nodes.

Simulation Pipeline Overview. Left: PettingZoo API integration, facilitating agent-simulation interaction via Python and RESTful requests for practical task offloading optimization. Right: network topology with worker nodes and connections, enabling task generation and state sharing for RL agent training.



(a) Reinforcement Learning Loop.

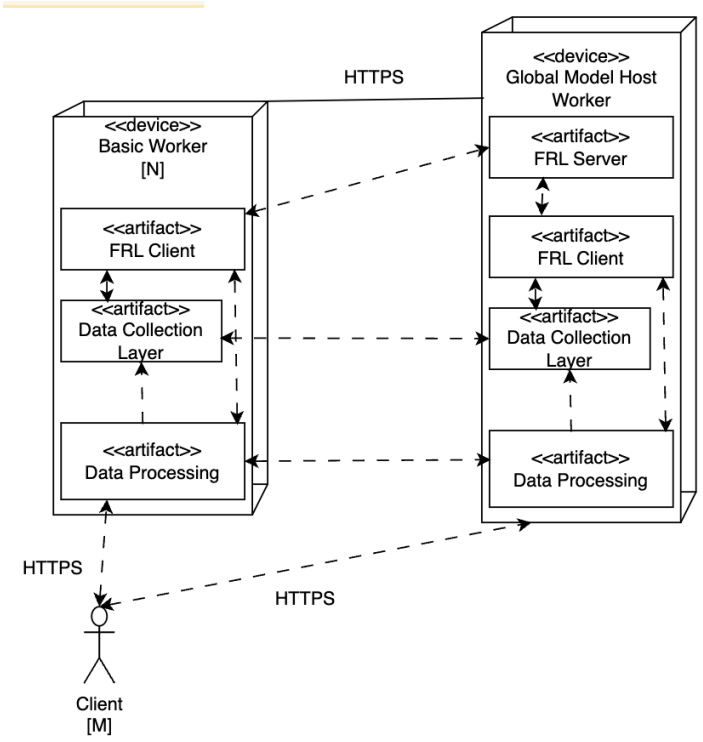
(b) Local state and action for a worker node.

FAUNO tool workflows

Diagram showing the deployment models

The diagram below shows the key components and their interactions. The key components are:

1. Client [M]: This entity represents the end user or device that initiates requests and interacts with the network services. It communicates with the network through HTTPS.
2. Basic Worker [N]: These are the distributed nodes within the network responsible for data processing and local learning.
3. Global Model Host Worker: This central entity coordinates the federated learning process and maintains the global model. This component is optional and not included in the P2P scenario.



FAUNO component interactions

The interactions between the key components are as follows:

HTTPS Communication: All interactions between the Client [M], Basic Workers [N], and the Global Model Host Worker are secured using HTTPS, ensuring data integrity and confidentiality during transmission.

Client to Basic Worker: The Client communicates with the Basic Workers via HTTPS to request network services. The Basic Workers process these requests and handle the necessary computations.

Basic Worker to Global Model Host Worker: Basic Workers interact with the Global Model Host Worker to participate in the federated learning process. They send local model updates to the FRL Server and receive the aggregated global model in return.

Diagram describing the modules composing the tool

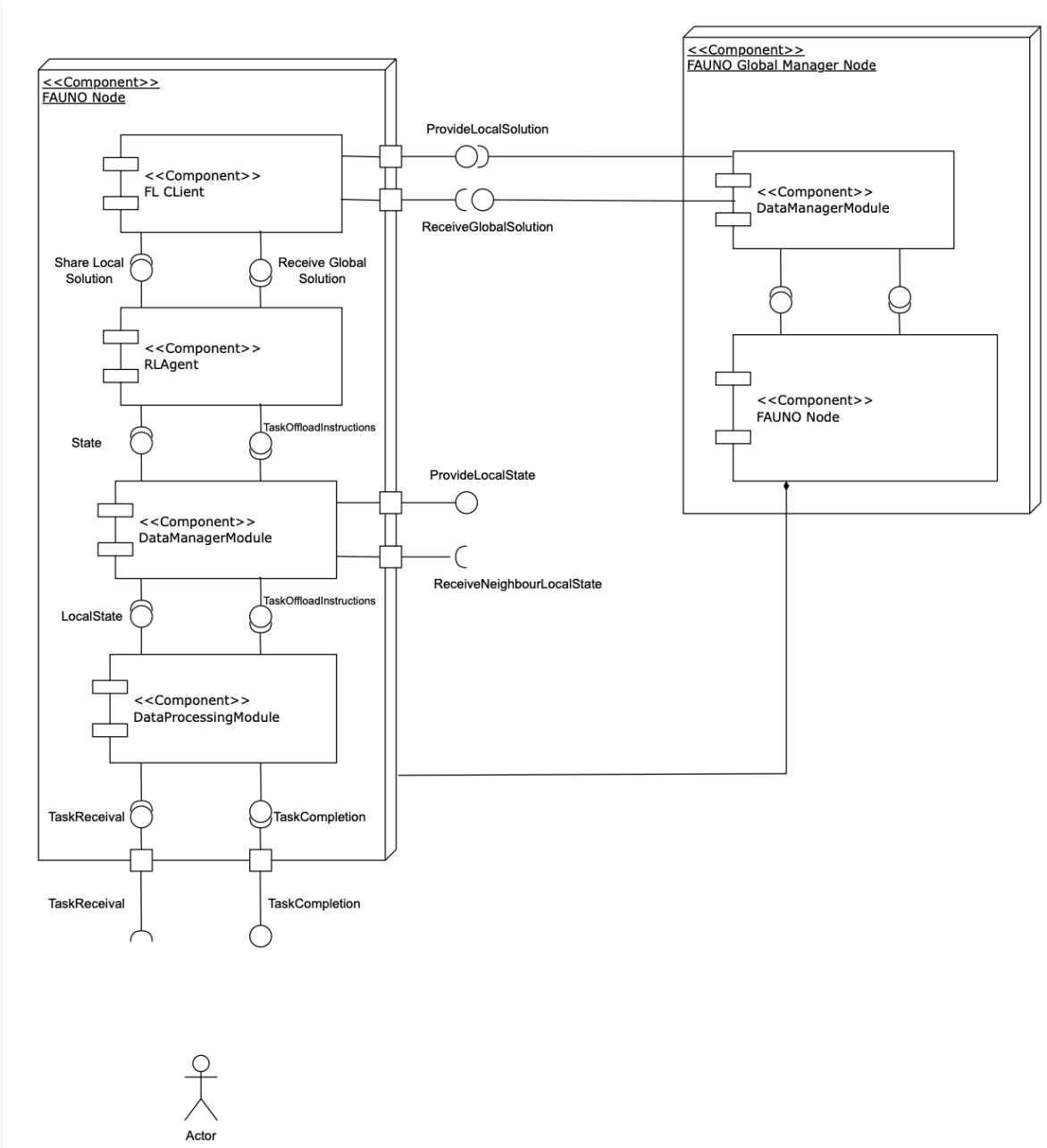
FAUNO comprises several interconnected modules, including:

- **Local Learning Agents**: Nodes equipped with reinforcement learning capabilities that operate based on local data and reward signals.

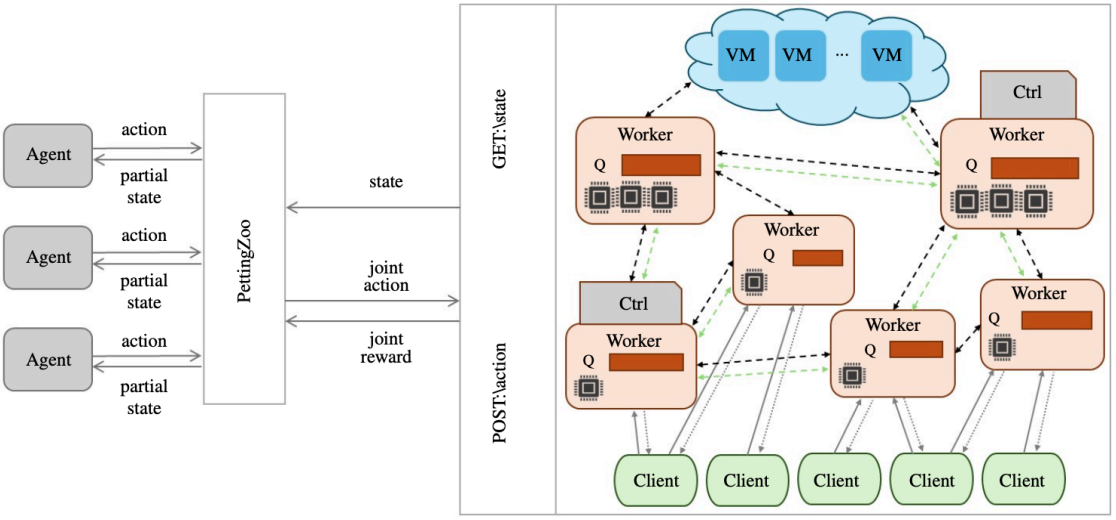
- **Federated Coordinator**: A central entity that aggregates local models and updates the global model without accessing raw data. Optional.

- **Communication Interface**: Facilitates secure and efficient data exchange between nodes and the federated coordinator. To be implemented.

Funded by the European Union



FAUNO modules



FAUNO interactions between agents and workers

The workflow of FAUNO can be summarised as follows:

1. **Data Collection**: Local nodes collect state information and task specifications.

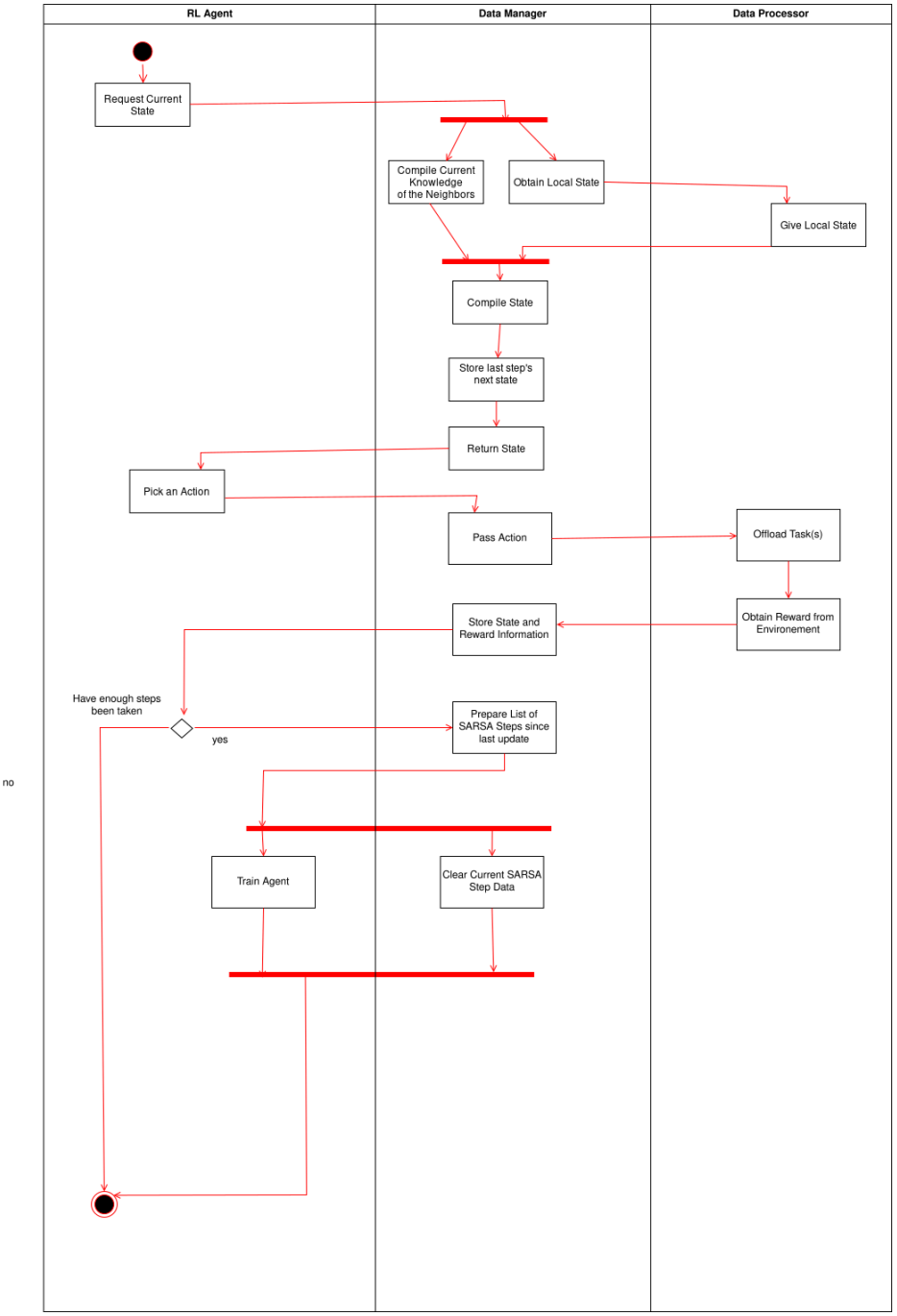
2. **Local Training**: Nodes perform reinforcement learning locally, generating local models and reward signals.

3. **Model Aggregation**: The federated coordinator aggregates the local models to update the global model.

4. **Policy Deployment**: Updated policies are deployed back to the local nodes for implementation.

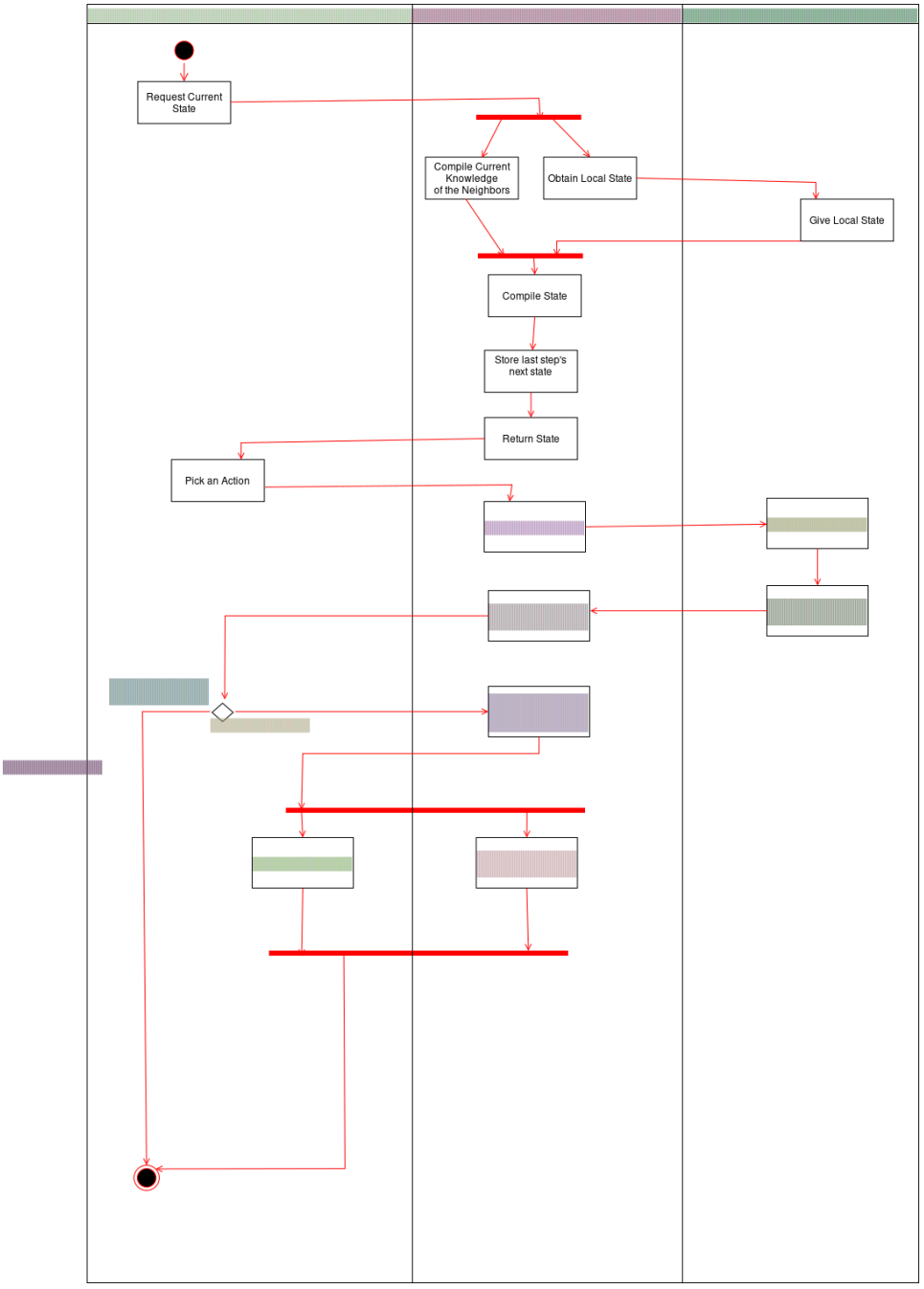
2.4.8.2 Tool Dynamic Behaviour

Following, there is a detailed timeline illustrating the interaction protocols between local nodes and the federated coordinator, highlighting key events such as data exchange, model updates, and policy deployment.



FAUNO dynamic behaviour for local training

The diagram above shows the local training cycle of the FAUNO Nodes. Each node will independently and without sharing data interact with its local view of the shared environment. The agent will execute a given number of steps with a given policy, and after a threshold is reached, it trains on those steps and clears its memory for the next batch of steps.



FAUNO dynamic behaviour for model sharing

As the diagram above displays, the FAUNO nodes will share what they learned with the FAUNO General Manager Node responsible for aggregating their results after a given number of local training steps.

2.4.8.3 Tool Operational details

Dependencies for FAUNO may include the following:

- Machine Learning Libraries: TensorFlow, PyTorch for implementing reinforcement learning models.
- Federated Learning Frameworks: Tools such as TensorFlow Federated, or Flower, for managing federated learning processes.

- Network Communication Protocols: Secure communication libraries to facilitate data exchange between nodes.

FAUNO can be operated through a command-line interface, with commands for initiating local training, aggregating models, and deploying policies. A configuration file abstracts technical complexities, allowing users to specify key parameters such as learning rates, reward functions, and model update frequencies.

The tool is under continuous development and testing, with a focus on enhancing scalability and robustness. Feedback from use cases will be incorporated to refine and optimise FAUNO's performance.

2.4.9 T-WP5-06 Early-Exit (Lightweight Functionality)

This tool provides a more lightweight version of a DNN model with multiple exits in order to minimise the inference latency (trade-off is the decrease in model accuracy). Specifically, the Early Exit tool transforms a trained Deep Neural Network (DNN) to an enhanced model variant equipped with several early exit opportunities. These exits facilitate a reduction in latency during the model's inference phase relative to the original DNN. The compromise, however, lies in an accuracy decrease. Additionally, a distributed form of Early Exit can be used, which leverages external resources beyond the local client or device, utilising edge computing or cloud-based services to further optimise performance.

The architecture of the DNN therefore incorporates multiple exit points, both during the training process and the inference stage. Consequently, the model is capable of delivering an output at the initial exit point, along with a corresponding confidence or accuracy metric. Should this confidence measure meet a predetermined threshold, the model opts for an early exit strategy. This approach conserves resources, including energy and computational power, and ensures minimal latency. It is particularly advantageous for classification tasks and is compatible with scenarios in TID and ACT use cases. The dependencies/Interactions with other tools include Tensorflow, Keras and Pytorch libraries are required for performing the training of the DNN model. No dependencies on other TaRDIS tools are foreseen at the moment.

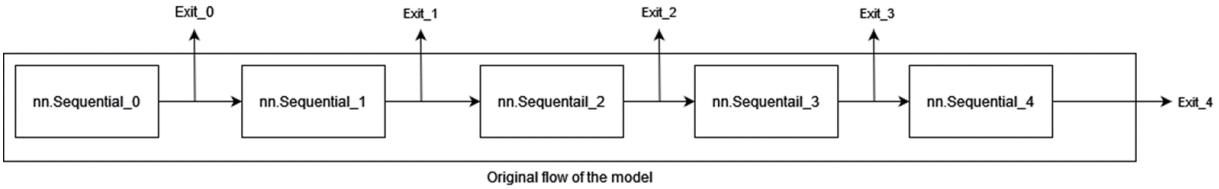
The input variables for this tool include the following parameters:

- the base ML model (for example .pt) that the developer wants to transform to its early-exit version.
- the type of the ML task that the base original model was trained for (classification or regression).
- the number of desired exits/outputs that shall be included in the new DNN architecture.
- the accuracy threshold that the early-exit version of the DNN model shall achieve in order to be reliable.
- the training dataset (or specification about the location of the training dataset) that the original model was trained upon is required for training also the early-exit version (labels are required at all the model exits).

The output of this tool is the new early-exit version of the DNN, according to the specified input parameters. The output can also include a metadata file with information related to the model accuracy, the training dataset, or the new architecture of the DNN (for instance where in the architecture the model exits are located).

2.4.9.1 Tool Architecture

The following figure demonstrates the whole DNN model with 4 exits (3 of them are early-exits). The parts between the exits are wrapped in an nn.Sequential() and the exits are automatically placed between these wrappers.



Early-Exit tool architecture

The workflow of the early-exit tool can be summarised in the following steps:

- Step 1: The developer (or another tool) initiates the process of lightweight training with the early-exit tool. In this context, the tool receives the required input variables, including the trained DNN model, the training dataset, the number of desired exits/outputs, the accuracy threshold and the type of the ML model.
- Step 2: The tool constructs the DNN with the new dimensionality (including the early exits at the specified locations) and conducts the training of the DNN, utilising the training dataset until the required accuracy threshold is reached.
- Step 3: The trained lightweight version of the DNN that includes the early exits is ready and can be used for inference purposes, exploiting also the early exits for latency decrease. Hence, the early-exit tool outputs the lightweight version of the ML model while the accuracy level is within limits.

2.4.9.2 Tool Persistence

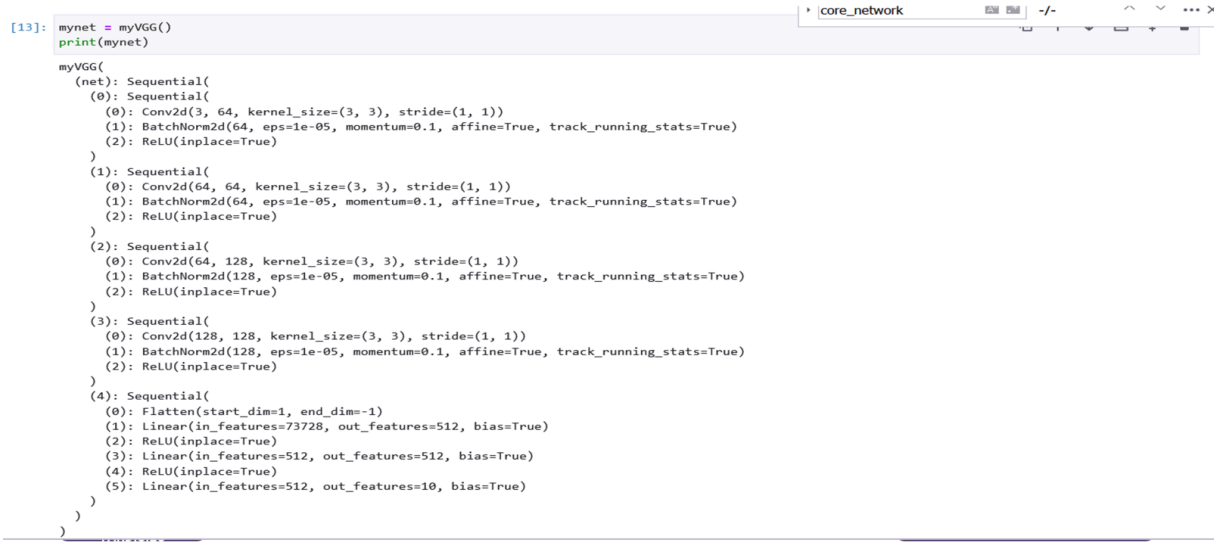
The training dataset that the original model was trained upon (or specification about the location of the training dataset in a data storage) is required for training the version of the model that includes the additional early exits.

2.4.9.3 Tool Operational details

Regarding the operational details for providing the tool functionalities, the following dependencies are required:

- Machine Learning Libraries: Tensorflow, Keras and Pytorch libraries are required for performing the training of the lightweight DNN model.
- Network Communication Protocols: Secure communication (e.g., libp2p) libraries are required to enable the data exchange between nodes in the feed-forward operation of the early-exit model.

The operation of the tool is through command-line by default. A DNN initial model example is shown below, separating the original DNN in 4 sequential networks:

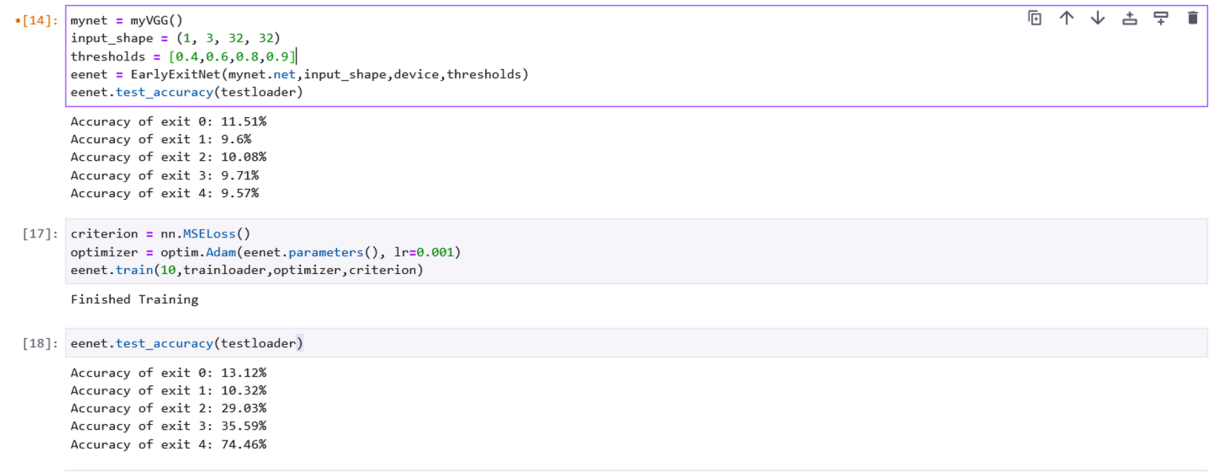


```
[13]: mynet = myVGG()
      print(mynet)

myVGG(
  (net): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (2): Sequential(
      (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (3): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (4): Sequential(
      (0): Flatten(start_dim=1, end_dim=-1)
      (1): Linear(in_features=73728, out_features=512, bias=True)
      (2): ReLU(inplace=True)
      (3): Linear(in_features=512, out_features=512, bias=True)
      (4): ReLU(inplace=True)
      (5): Linear(in_features=512, out_features=10, bias=True)
    )
  )
)
```

Example of a constructed Neural Network that includes early-exits

Furthermore, the construction and training process of the early exit model, including the input shape dimensions and the accuracy thresholds are depicted in the following figure. Please also note that the accuracy of all the exits is similar during initialization, whereas the accuracy of higher-hierarchy exits increases when the model is trained.



```
[14]: mynet = myVGG()
      input_shape = (1, 3, 32, 32)
      thresholds = [0.4,0.6,0.8,0.9]
      eenet = EarlyExitNet(mynet.net,input_shape,device,thresholds)
      eenet.test_accuracy(testloader)

Accuracy of exit 0: 11.51%
Accuracy of exit 1: 9.6%
Accuracy of exit 2: 10.08%
Accuracy of exit 3: 9.71%
Accuracy of exit 4: 9.57%

[17]: criterion = nn.MSELoss()
      optimizer = optim.Adam(eenet.parameters(), lr=0.001)
      eenet.train(10,trainloader,optimizer,criterion)

Finished Training

[18]: eenet.test_accuracy(testloader)

Accuracy of exit 0: 13.12%
Accuracy of exit 1: 10.32%
Accuracy of exit 2: 29.03%
Accuracy of exit 3: 35.59%
Accuracy of exit 4: 74.46%
```

Example of the early-exit tool functionality, illustrating the performance of the multiple exits during the training process

2.4.10 T-WP5-07 Knowledge Distillation (Lightweight Functionality)

The Knowledge distillation tool aims to provide a more lightweight version of the DNN in terms of NN complexity, hidden layers and number of neurons. Knowledge distillation is a process where a larger, more complex network (referred to as the "teacher" network) is used to train a more compact "student" model. The essence of this technique is the transfer of knowledge from the teacher to the student. This method is particularly beneficial in classification scenarios where rigid class labels could lead to confusion between similar classes with overlapping characteristics. Through knowledge distillation, "soft labels" are generated, which reduce the penalization for common features between classes. This technique also enables the student model to be trained without needing the original dataset. The primary benefit of knowledge distillation lies in the student model's reduced size, making

it more resource-efficient than the teacher model, thus conserving energy and computational power. While knowledge distillation can also be adapted for regression tasks by adjusting the loss function, it is most effective in classification problems.

Knowledge Distillation entails the transfer of expertise from a large DNN, known as the teacher, to a smaller, more efficient model, known as the student. This technique conserves energy and computational resources while maintaining satisfactory predictive accuracy within the DNN. While applicable to both classification and regression tasks, Knowledge Distillation performs best in classification contexts, offering superior results. Consequently, it is associated with use cases such as ACT, TID, and potentially EDP. Dependencies/Interactions with other tools includeTensorflow, Keras and Pytorch libraries are required for performing the training of the DNN model. No dependencies on other TaRDIS tools are foreseen at the moment.

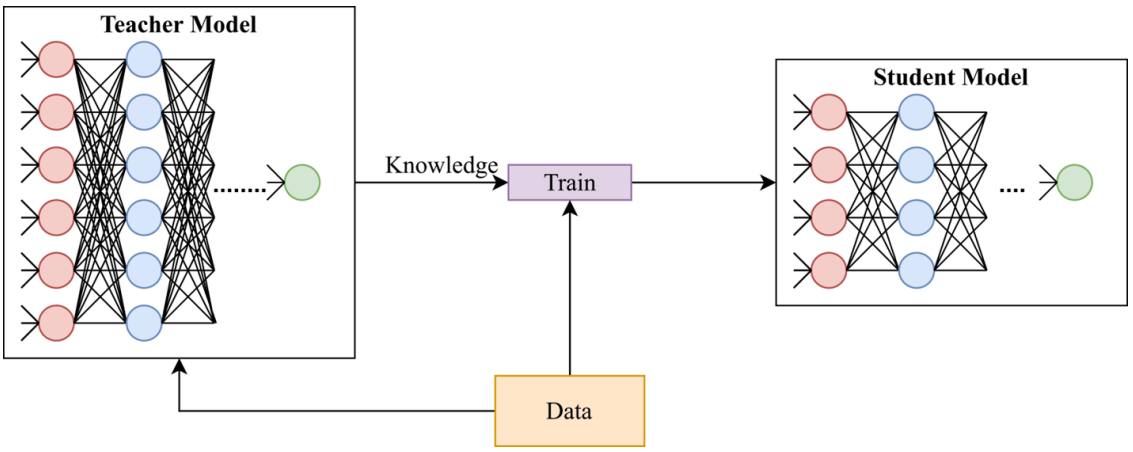
The inputs of the KD tool are the following:

- The base teacher ML model (e.g., in a .pt format) that the ML developer wants to transform to a more lightweight version by using the KD technique.
- The type of the ML task that the model was originally trained for (classification or regression task).
- The ML developer can select a loss function for the training process of the student network (amongst several options). We will provide a default option for ease of use.
- The ML developer shall provide as input the number of hidden layers required for the student model, or alternatively the lightweight rate of the output model. In this way, the tool will have as input the number of hidden layers that the student model will be built upon compared to the teacher model.
- The accuracy threshold is required, since it is the stopping criterion for the training process. In essence, the knowledge gained from the teacher in the student network is a continuous process until the accuracy threshold is reached.
- The training dataset that the original teacher model was trained upon (or the location of the training dataset in a data storage) is required for training purposes of the student network.

The output of the KD tool is the more lightweight version of the DNN model (student model). The output can also include a metadata file with information related to the achieved accuracy, the training dataset, and the lightweight percentage of the new model.

2.4.10.1 Tool Architecture

The internal architecture of the KD tool is illustrated in the following figure. The student model is initialised, based on the specifications and dimensionalities provided by the user and the training process is initiated, i.e., the training experience of the teacher model is distilled in the student model.



Knowledge Distillation tool architecture

The flow functionality of the KD tool can be summarised in the following steps:

- Step 1: the KD training process is initiated by the developer/user or by another tool, by specifying the dimensionality of the student network and providing the required input parameters (accuracy threshold, type of ML task, loss function), including the base teacher model (for instance, in a .pt format) and the training dataset.
- Step 2: once the tool has all the required inputs, the training process can be initiated and the procedure of knowledge distillation between the teacher and the student model can take place until the required level of accuracy is reached.
- Step 3: once the training is completed and the required accuracy is achieved, the KD tool outputs to the developer the lightweight version of the input model, i.e., the student model.

2.4.10.2 Tool Persistence

The training dataset that the original model was trained upon (or specification about the location of the training dataset in a data storage) is required for training the lightweight version of the model.

2.4.10.3 Tool Operational details

The interface is command-line by default and the following dependencies are required for the operation of the KD tool:

- Machine Learning Libraries: Tensorflow, Keras and Pytorch libraries are required for performing the training of the student model.

2.4.11 T-WP5-08 Pruning (Lightweight Functionality)

This tool provides a more lightweight version of the input DNN by using the pruning method. Pruning involves adjusting a network's weights to zero, which can be executed in an unstructured manner—targeting individual weights based on certain criteria—or in a structured way—eliminating entire neurons and their associated connections. This process requires a deliberate strategy; while the 'brain surgeon' method is recognized as optimal, it is also notably complex to implement. For the TaRDIS use cases, a specialised pruning technique has been formulated, and efforts to integrate it into Python libraries are currently in progress.

Pruning is the process of setting certain neuron weights in a DNN to zero, particularly those that have minimal influence on the model's performance. This technique streamlines the

inference phase, leading to faster processing and conservation of energy and computational resources. Given its effectiveness across both classification and regression tasks, pruning is applicable to a variety of use cases, including TID, EDP, GMV, and ACT.

The dependencies/Interactions with other tools include Tensorflow, Keras and Pytorch libraries are required for performing the training of the DNN model. No dependencies on other TaRDIS tools are foreseen at the moment.

The inputs for the tool consist of:

- The base original DNN model (for instance, in a .pt format) that the ML developer wants to transform to a more lightweight version by fundamentally using the pruning technique.
- The compression rate (%) of the new version of the DDN, and the accuracy threshold (%) that the pruned version of the DNN model shall achieve. The compression rate is also called the sparse ratio of the DNN.
- The method that will be utilised for pruning, by selecting amongst several available options (for instance, structured or unstructured pruning). We will provide default selections to the ML developer
- The training dataset or the location of the dataset in a data storage, since the updated version of the DNN model needs to be trained against the same dataset as the original unpruned model.
- The output of the pruning tool is the more lightweight, pruned version of the original DNN model. The output can also include a metadata file with information related to the compression rate and the lightweight percentage of the new model.

2.4.11.1 Tool Architecture

In the following figure, the functionality of the pruning process is demonstrated. The pruning tool receives as input a pre-trained DNN model and performs the pruning process, providing a more lightweight version of the original model.

Pruning tool architecture

The flow functionality of the pruning tool can be summarised in the following steps:

- Step 1: The ML developer or another tool initiates the process of the pruning tool by providing the original DNN model, as well as the input variables required by the tool. These include the sparse ratio, the required accuracy of the pruned model, the training dataset, the pruning method, etc.
- Step 2: The actual training of the lightweight DNN model is conducted until the required levels of accuracy for the requested sparse ratio are achieved.

- Step 3: Once the pruned version of the original model has been successfully trained, it is provided back to the user as output of the tool.

2.4.11.2 Tool Persistence

The training dataset that the original model was trained upon (or specification about the location of the training dataset in a data storage) is required for training the lightweight version of the model.

2.4.11.3 Tool Operational details

The interface of the pruning tool is through command-line by default and the following dependencies are required for its appropriate operation:

- Machine Learning Libraries: TensorFlow, PyTorch for implementing the lightweight version of the DNNs and the training process of the pruned model.
- A wrapper function has been developed that abstracts all the technical knowledge from the user, requiring from him only the model, the shape of the input and the sparse ratio. Our function has been tested on simple linear networks, convolutional, as well as networks with parallel layers.

A simple example of the operation of the pruning tool is shown in the following figures. Initially, the call of the pruning function includes the original model, the sparse ratio (pruning rate) and the dimensions of the model.

```
net = myVGG()
input_shape = (1, 3,32,32)
pruned_model = prune_model(net, 0.3, input_shape)
✓ 4.3s                                                    Python
```

Example of the pruning tool function, including the original model and its dimensions, as well as the sparse ratio

Furthermore, the original model dimensions (including the output layer) is compared to the pruned model respective dimensions in the following figures, showcasing that the input features of the lightweight pruned model have been reduced according to the parameters specified by the user.

```
...
    (4): ReLU(inplace=True)
    (5): Linear(in_features=512, out_features=10, bias=True)
  )
```

```
...
    (4): ReLU(inplace=True)
    (5): Linear(in_features=359, out_features=10, bias=True)
  )
```

Example of the pruning tool functionality, comparing the input feature dimensionality between the original and the pruned model

2.4.12 T-WP5-09 Decentralised Federated Learning Framework (Fedra)

This tool provides an implementation of a decentralised federated learning framework integrated with p2p communications between the participating nodes. Fedra revolutionises federated learning by leveraging libp2p for a decentralised, anonymous peer-to-peer

communication framework. It allows for secure model weight exchange, upholding privacy and data ownership. Fedra tackles traditional federated learning challenges head-on by enhancing privacy, streamlining communications, and simplifying data management across varied and distributed data sources. Fedra provides the following capabilities:

- **Model Agnosticism**: Fedra framework is designed to be model-agnostic, supporting a wide array of machine learning models to suit various data types and learning tasks.
- **Decentralised Learning**: By leveraging a peer-to-peer network, Fedra ensures that learning is truly decentralised, offering enhanced privacy and autonomy over data.
- **Scalability and Flexibility**: Fedra is built to scale, accommodating an increasing number of nodes seamlessly, and allows for flexible network configurations to optimise performance.
- **Privacy by Design**: Privacy is at the core of Fedra's approach, ensuring that data remains on local nodes, mitigating the risks associated with central data collection.
- **Efficiency in Communication**: With libp2p, Fedra optimises communication protocols, resulting in reduced overhead and improved overall training efficiency.
- **Simplified Configuration**: Fedra streamlines setup through a single configuration file, node.conf, encapsulating all necessary information for the decentralised federated learning process, ensuring an efficient and user-friendly initialization for each peer in the network.

The decentralised federated framework can be potentially employed to train any ML model in a federated manner, since Fedra is model-agnostic. ML models for anomaly detection, regression, forecasting or even based on reinforcement learning can be effortlessly adapted in the Fedra framework. In this context, the Fedra framework is applicable to a variety of use cases, including TID, EDP, GMV, and ACT. Tensorflow, Keras and Pytorch libraries are required for performing the training of the DNN models. Moreover, libp2p is required for the node communication, while the flower framework is used to adopt the federated learning strategies. No dependencies on other TaRDIS tools are foreseen at the moment.

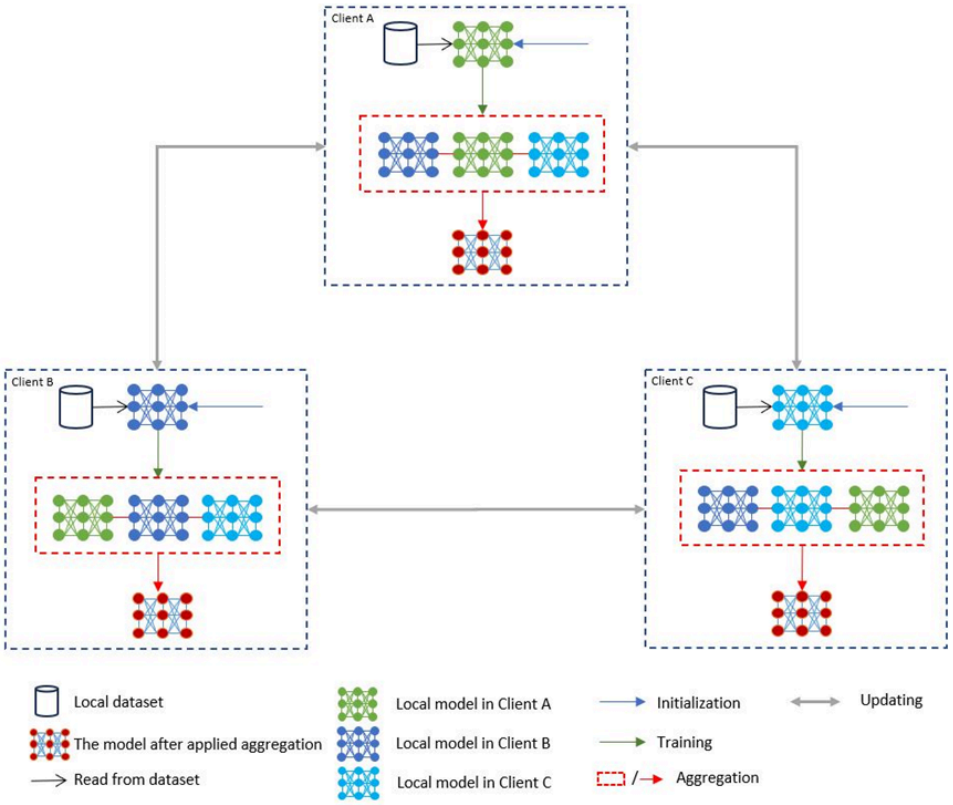
The inputs for the tool include:

- Fedra framework requires node-specific configurations, managed via a simple configuration file (node.conf). This file encapsulates essential parameters such as model details, training rounds, and P2P network settings, ensuring a seamless initiation into the federated learning process.
- Fedra also inherently considers that each node has access to its own training dataset that is stored locally in a database.
- The output of the Fedra tool is the trained ML model at each node participating in the federation process, as well as the performance of the training process (Fedra can be used to visualise the accuracy obtained with the training and testing datasets).

2.4.12.1 Tool Architecture

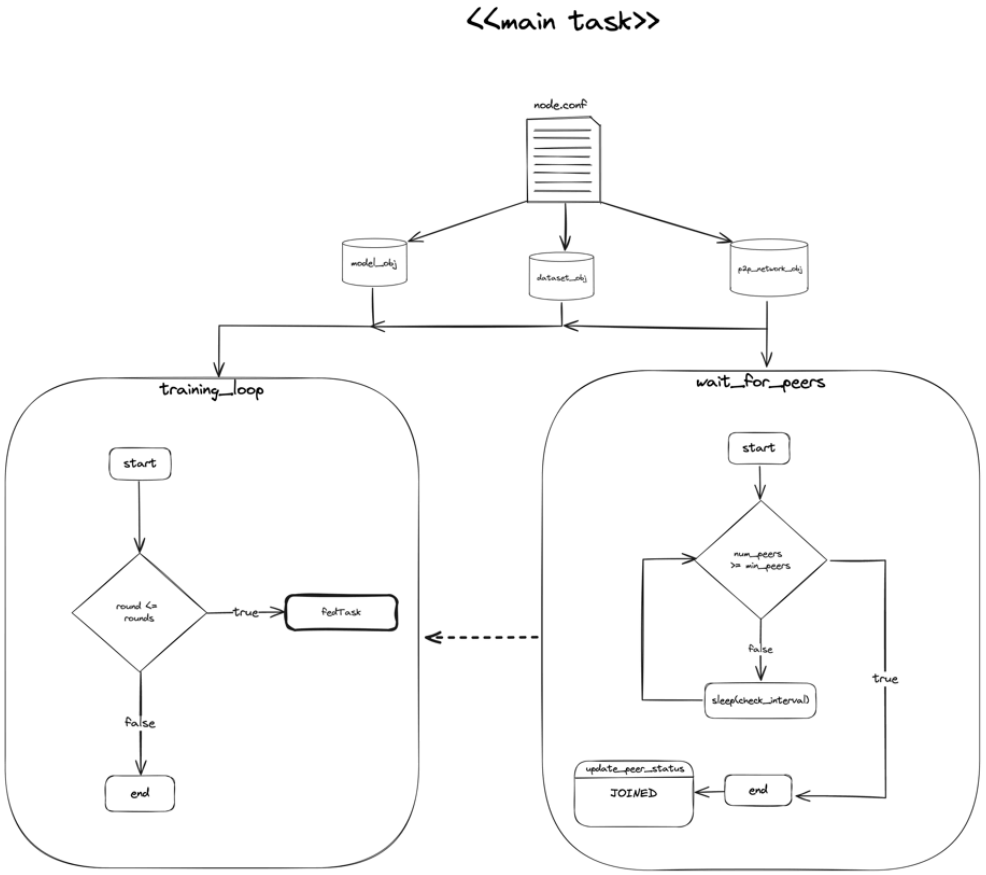
The following figure illustrates the decentralised federated learning process when three clients are considered in the framework. Each Client has its own ML model that is trained locally with the individual training dataset. Moreover, each client also periodically acknowledges the weights of its own model to the other clients participating in the framework in order to enable the federation process, i.e., the model aggregation (for instance, averaging of the weights in the FedAvg algorithm).

Funded by the European Union

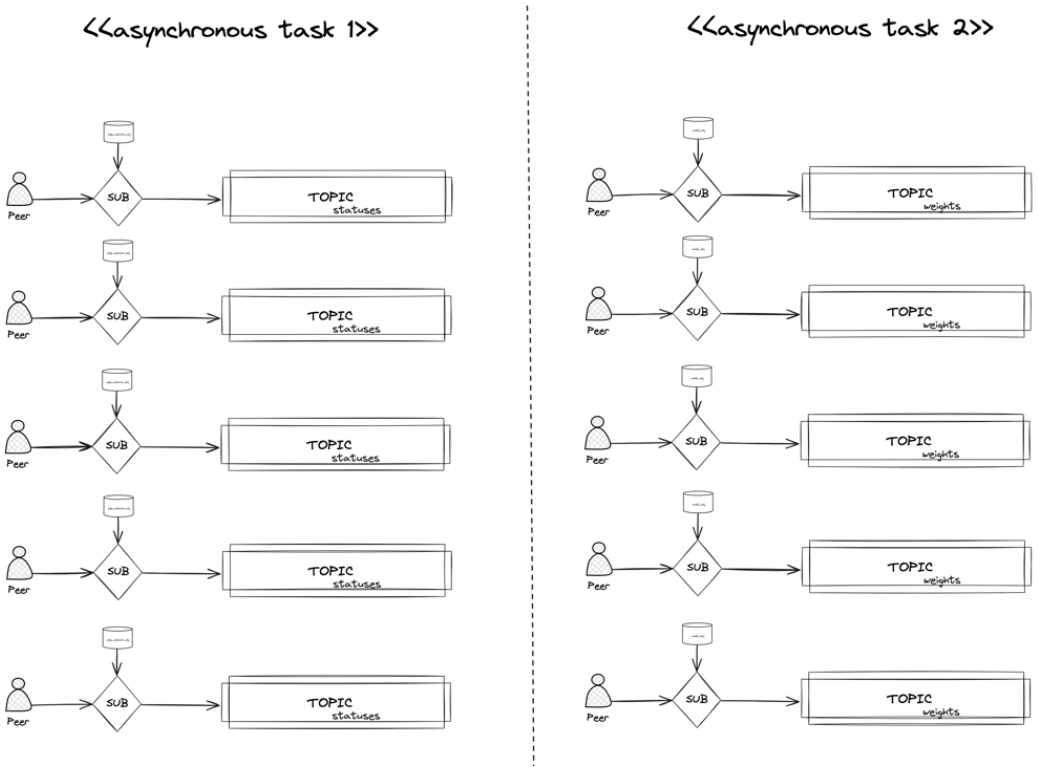


Fedra tool architecture

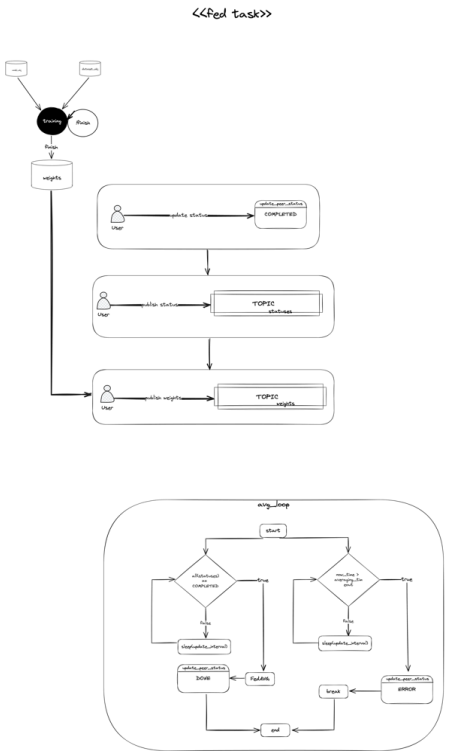
The processes involved in the development of Fedra are shown in the following figures, including the wait for peers workflow, the main training workflow, the asynchronous tasks that are used for publishing the statuses and the model weights of each node to facilitate the communication and the model exchange, the federated learning main task that includes the local training, the publishing of the model weights and the federation loop, as well as the shared objects amongst the participating nodes.



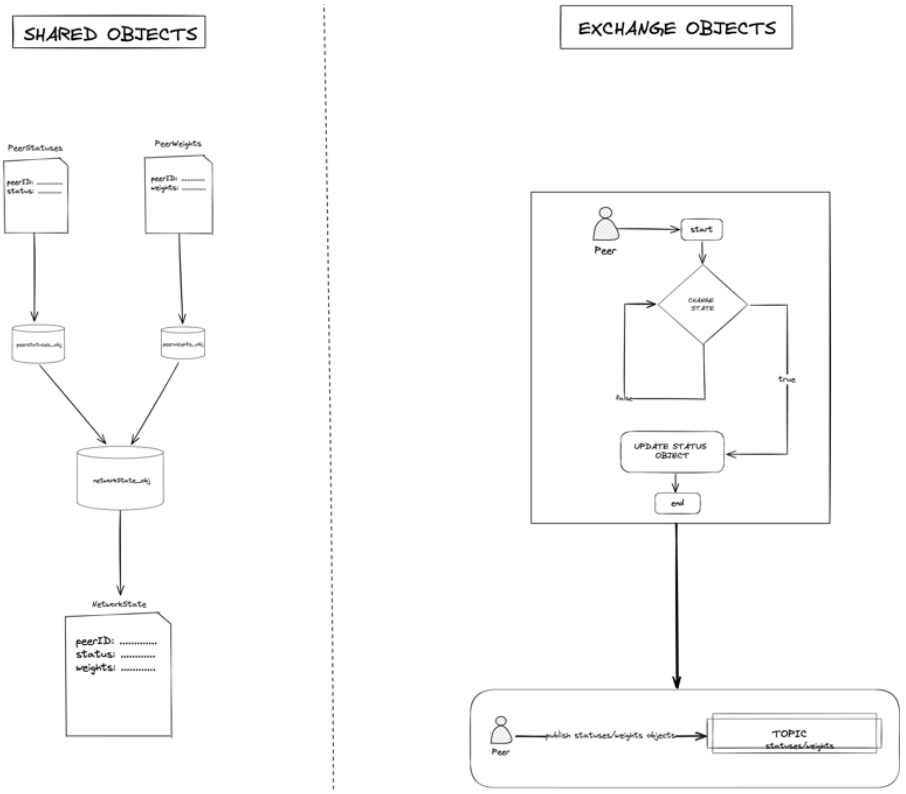
The main workflow of Fedra, including the wait for peers routine and the training loop



The two asynchronous tasks that are executed in the Fedra workflow, concerning the status acknowledgement and the weights exchange amongst the peers



Workflow of the federated learning that is executed during the training process by the peers that participate in the Fedra framework



Description of the objects that are shared and exchanged between the nodes in the Fedra framework

The flow functionality of the Fedra framework can be summarised in the following steps:

- Step 1: Initialization: The global model is initialised, and configuration parameters are distributed to all participating nodes.
- Step 2: Local Training: Each node trains the model locally with its dataset.
- Step 3: Model Update Exchange: Post-training, nodes serialise and exchange model updates amongst themselves via P2P communication.
- Step 4: Local Aggregation: Each node deserializes received updates and performs local federated averaging to refine their model.
- Step 5: Convergence and Repetition: The iterative process continues, with each round of training and aggregation progressively enhancing the model's accuracy.
- Diagram showing the extension points, APIs etc.

2.4.12.2 Tool Operational details

The interface of Fedra is through command-line by default and the following points are required for its operation:

- Clone the repository: git clone https://github.com/anaskalt/fedra.git
- Install the required dependencies: pip install -r requirements.txt
- Configure your node by editing the node.conf file with your specific settings.
- Run Fedra: python3 fedra.py

2.5 WP6: Data Management and Distribution Primitives

Work Package 6 (WP6) is responsible for designing and developing the runtime support for TaRDIS heterogeneous swarms. Evidently this translates to a myriad of different tools that can be used by developers for different purposes during their design and development.

The work package provides fundamental abstractions for swarm membership tracking (mostly based in overlay networks, where swarm elements cooperate to distribute load (and information) regarding the elements currently in the swarm; and communication abstractions from point-to-point to point-to-multipoint communication primitives, the latter usually achieved through cooperation among swarm elements (Task 6.1). Storage abstractions for the swarm including storage systems that have dedicated infrastructure and lifecycles independent of the swarm elements, which is relevant for durability and archiving; and integrated swarm storage abstractions that are directly maintained by swarm elements (Task 6.2); and telemetry acquisition and self-management mechanisms for the swarm, that allow to collect runtime information about the swarm elements and their execution environment towards guiding the on-line self-reconfiguration of the swarm (Task 6.3).

Furthermore, WP6 is also responsible for providing connectors to existing solutions, services, or abstractions commonly used in building distributed systems as a way to facilitate the integration of TaRDIS technology into legacy systems.

This translates to different types of tools, that in more detail, can be (in general) separated into the classes discussed in the following.

2.5.1 Runtime Support Tools

This is the fundamental support that on one hand provides a development support but also runtime functionalities to build swarm applications by controlling the lifecycle of some of the other tools reported here.

T-WP6-03: Actyx: Reliable event broadcast with configurable durability
T-WP6-04: Babel (extended to support autonomous swarms)

WP6 runtime support requirements

requirement ID and name	T-WP6-03	T-WP6-04
RF-WP6-G-02 support for (Android) mobile clients	✔	✔
RF-WP6-CP-20 isolation between different communication abstractions	✔	✔
RF-WP6-SA-28 log-based storage system with eventual consistency	✔	
RF-WP6-SA-31 isolation between data segments within a storage abstraction	✔	✔
RF-WP6-TA-39 scalable telemetry acquisition mechanisms	✔	✔
RF-WP6-CM-42 replication of essential agents	✔	✔

2.5.2 Common-APIs

These are libraries that specify common mechanisms to interact with different classes of the tools developed in the context of TaRDIS. This has the important role of providing some decoupling between different components which is relevant to allow developers to easily swap a particular swarm tool/component by another one that provides the same functionality or service for the swarm, ensuring high modularity for the TaRDIS swarm architecture.

T-WP6-01: Generic API for Decentralised Overlay & Communication Protocols

T-WP6-11: Babel Common APIs for Adaptive Protocols

WP6 common API requirements

requirement ID and name	**T-WP6-01**	**T-WP6-11**
RF-WP6-MA-05 dynamic self-managed overlay networks	✔	✔
RF-WP6-MA-06 biased dynamic self-managed membership abstractions	✔	
RF-WP6-MA-07 location-aware dynamic self-managed membership abstractions	✔	✔
RF-WP6-MA-08 isolation between different membership abstractions	✔	✔
RF-WP6-MA-09 hierarchical dynamic self-managed membership abstractions	✔	✔
RF-WP6-MA-10 cluster-based dynamic self-managed membership abstractions	✔	✔
RF-WP3-API-01 logging and monitoring		✔
RF-WP3-API-02 reconfiguration capabilities		✔

2.5.3 Membership Protocols

These are a set of different decentralised membership protocols that are a fundamental building block for achieving autonomous (and heterogeneous swarms), allowing elements of the swarm to have local (many times to ensure scalability, partial) membership information of the swarm (i.e., information about other participants in the swarm). Different membership services might provide different properties (e.g., global connectivity) or even services (e.g., peer sampling or application-level routing).

T-WP6-02: An Epidemic and Scalable Global Membership Service

T-WP6-12: Decentralised Membership Protocols for Swarms

WP6 membership service requirements

requirement ID and name	T-WP6-02	T-WP6-12
RF-WP6-G-01 management of cryptographic material by participant	✔	✔
RF-WP6-MA-04 authenticated decentralised membership abstractions	✔	✔
RF-WP6-MA-05 dynamic self-managed overlay networks		✔
RF-WP6-MA-06 biased dynamic self-managed membership abstractions		✔
RF-WP6-MA-07 location-aware dynamic self-managed membership abstractions		✔
RF-WP6-MA-08 isolation between different membership abstractions	✔	✔
RF-WP6-MA-09 hierarchical dynamic self-managed membership abstractions		✔
RF-WP6-MA-10 cluster-based dynamic self-managed membership abstractions		✔
RF-WP6-MA-11 administrative domain clusters emerge from dynamic self-managed membership abstractions		✔
RF-WP6-MA-12 local global information about active elements in the swarm	✔	
RF-WP6-MA-14 scalable decentralised membership abstractions	✔	✔
RF-WP6-MA-15 always available membership abstractions	✔	✔
RF-WP6-TA-36 telemetry acquisition should include membership information	✔	✔

2.5.4 Communication Protocols

These are a set of distributed (and many times decentralised) communication protocols that can provide communication services with different properties to elements of the swarm, over

which more powerful abstractions and services can be built. Different communication protocols offer different services (e.g., broadcast or publish-subscribe), guarantees (reliable delivery or best-effort delivery), or properties (e.g., authentication or integrity).

T-WP6-13: Decentralised Communication Protocols for Swarm

WP6 communication protocol requirements

requirement ID and name	T-WP6-13
RF-WP6-CP-16 point-to-point secure communication primitives	✔
RF-WP6-CP-17 point-to-multipoint secure communication primitives	✔
RF-WP6-CP-18 communication primitives that provide privacy	✔
RF-WP6-CP-19 reliable point-to-multipoint communication primitives	✔
RF-WP6-CP-20 isolation between different communication abstractions	✔
RF-WP6-CP-21 real-time compatible point-to-multipoint communication primitives	✔
RF-WP6-CP-22 real-time compatible point-to-point communication primitives	✔
RF-WP6-CP-23 scalable decentralised point-to-multipoint communication primitives	✔
RF-WP6-CP-24 always available point-to-multipoint communication primitives	✔
RF-WP6-TA-35 durability of communication for auditing	✔

2.5.5 Data Management Services

Data management services allow management of application data. They provide swarm elements the ability to access or modify shared data among them. With different services providing different semantics or guarantees.

T-WP6-03: Actyx: Reliable event broadcast with configurable durability
T-WP6-05: Arboreal: Cloud–Edge Data Management, Dynamic Replication
T-WP6-06: PotionDB: Strong Eventual Consistency under Partial Replication

WP6 data management requirements

requirement ID and name	T-WP6-03	T-WP6-05	T-WP6-06
RF-WP6-SA-25 durable and non-forgeable storage service	✔	✔	✔
RF-WP6-SA-26 isolation between applications using same storage abstractions		✔	✔
RF-WP6-SA-27 federated learning participants state must be managed	✔	✔	
RF-WP6-SA-28 log-based storage system with eventual consistency	✔		
RF-WP6-SA-29 available and durable blob-based storage system	✔		
RF-WP6-SA-30 distributed data storage abstraction exposes telemetry information	✔	✔	✔
RF-WP6-SA-31 isolation between data segments within a storage abstraction		✔	✔
RF-WP6-SA-32 decentralised storage solutions must be scalable	✔	✔	✔
RF-WP6-SA-33 always available distributed storage abstractions	✔	✔	✔
RF-WP6-TA-35 durability of communication for auditing	✔	✔	
RF-WP6-TA-37 telemetry acquisition should include storage cost for data management nodes		✔	✔
RF-WP6-CM-43 configure data retention and replication on distributed data management systems	✔		✔
RNF-WP4-PROP-02 data management and replication		✔	✔
RNF-WP4-VER-02 distributed data management		✔	✔

2.5.6 Telemetry Acquisition Services

Telemetry acquisition services allow the collection of runtime information on the swarm execution (e.g., resource consumption or internal components statistics) and about their

execution environment. This data might need to be shipped to different locations of the swarm, or even external components of the swarm for analysis (e.g., machine learning training).

T-WP6-09: Telemetry Acquisition for Decentralised Systems for Containers
T-WP6-10: Telemetry Acquisition for Decentralised Systems in Babel

WP6 telemetry acquisition requirements

requirement ID and name	T-WP6-09	T-WP6-10
RF-WP6-SA-30 distributed data storage abstraction exposes telemetry information	✔	✔
RF-WP6-TA-34 monitorization of memory and communication for application components	✔	✔
RF-WP6-TA-35 durability of communication for auditing		✔
RF-WP6-TA-36 telemetry acquisition should include membership information		✔
RF-WP6-TA-37 telemetry acquisition should include storage cost for data management nodes	✔	✔
RF-WP6-TA-38 monitorization of metrics to support training of self-management	✔	✔
RF-WP6-TA-39 scalable telemetry acquisition mechanisms	✔	✔
RF-WP6-TA-40 always available telemetry abstraction	✔	✔
RF-WP3-API-01 logging and monitoring	✔	✔

2.5.7 Self-Management Support

These tools are tailored to guide, through different means, online reconfiguration of swarm components or even elements to ensure that the swarm remains correct despite changes on its execution environment, size, or even workloads, being a key aspect to ensure that the swarm is highly autonomous and adaptive to new situations.

T-WP6-08: Distributed Management of Configuration based on Namespaces
T-WP6-14: Decentralised Estimator of Swarm size
T-WP6-15: Localised Simple Static Autonomic Swarm Manager

WP6 self-management requirements

requirement ID and name	T-WP6-08	T-WP6-14	T-WP6-15
RF-WP6-MA-13 swarm self-configuration	✔	✔	✔
RF-WP6-SA-27 federated learning participants state must be managed			✔
RF-WP6-TA-38 monitorization of metrics to support training of self-management		✔	
RF-WP6-CM-41 dynamically adapt the memory consumption and communication patterns of application components	✔		✔
RF-WP6-CM-42 replication of essential agents	✔		✔
RF-WP6-CM-43 configure data retention and replication on distributed data management systems			✔
RF-WP6-CM-44 always available configuration management	✔		✔
RF-WP3-MOD-05 specifying device capabilities	✔		✔
RF-WP3-API-02 reconfiguration capabilities	✔		✔

2.5.8 Adaptors to External Services

Adaptors to external services provide a way to interact with legacy services or systems using an API akin to the one provided by other TaRDIS tools, ensuring uniformity in access, and making it easier to introduce TaRDIS components to legacy systems, or the interaction of TaRDIS swarms with external services.

Many of the tools listed in this section are detailed in D6.1 «Report on the first iteration of TaRDIS toolbox components», however some have been developed since the delivery of D6.1 and will only be presented in detail on the upcoming D6.2.

T-WP6-07: Integration of Storage Solutions into the TaRDIS ecosystem

WP6 external adapter requirements

requirement ID and name	T-WP6-07
RF-WP6-G-03 exactly once external adaptors	✔

requirement ID and name	T-WP6-07
RF-WP3-API-03 interfacing with external services	✔

2.5.9 T-WP6-01 Generic API for Decentralised Overlay & Communication Protocols

In deliverable D6.1 we have presented our efforts in designing a Generic API for supporting interactions with decentralised overlays, which constitute the fundamental materialisation of scalable membership abstractions for decentralised systems, and communication protocols for this setting. This generic API was built on top of the Babel framework (which is presented alongside a summary of its current evolution within TaRDIS further ahead in section 2.5.12). The fundamental idea of this Generic API was to, on one hand, abstract the different functionalities/services that these types of distributed protocols can provide to swarm applications, while at the same time offering a flexible interaction model through different types of APIs for these abstractions.

This was achieved by identifying a set of fundamental services, that independently of their implementation, can be interacted with through a common interface. In detail, this generic API focused on the following abstractions:

2.5.9.1 Membership Management Service

This type of service captures abstractions that provide some form of network management operations that can be leveraged by different decentralised protocols and applications to gather information about the membership of the system. In particular this is an interface that should be compatible with most (virtually all) decentralised overlay management protocols, independently of the type of overlay being managed by them (unstructured or random versus structured, commonly distributed hash tables (DHTs)).

Natural functionalities of such service includes mechanisms to allow a swarm element (i.e., some process that runs on a physical machine) to join the swarm system, becoming known to some of the other elements currently active in the swarm, and becoming aware of some other elements in the swarm. This leads this abstraction to provide also a mechanism for a swarm element to obtain information about logical neighbours in the swarm, or a random sample of other swarm elements.

The API of this service includes the following requests (some of which generate replies and notifications[31].

Requests:

Join(Set<Host> contacts) The Join operation is responsible for allowing a node to join a network given a set of process identifiers (defined here with the Host data type)

Leave() The Leave operation allows a node to leave the network. This operation does not require any parameters and protocols can either rely on an implementation that executes tasks to gracefully leave the system.

[31] To better understand the difference between these types of events, the reader can see their brief description under inter-protocol communication under Babel in Section 2.5.12.2.

GetNeighborsSample(byte[] requestID, Integer t) → (byte[] requestID, Set<Host> sample) The GetNeighborsSample operation allows to obtain from the membership management service a set of process identifiers that are part of the system.

Notifications:

NeighborUp(Host h) We considered the NeighborUp notification to be asynchronous triggered by a protocol when a new peer becomes known,

NeighborDown(Host h) We considered the NeighborDown notification to be triggered by protocols that implement the membership management service interface when a process in the system is detected as no longer being available,

2.5.9.2 Application-level Routing Service

The Routing service interface provides the operations related with the capability of obtaining a node, or a set of nodes, active in the swarm, based on the proximity of their logical identifier (typically a random bitstring) to some other identifier, therefore effectively allowing routing information or requests to a particular node.

This interface can be exposed by any distributed protocol that materialises a distributed hash table (DHT) or that supports some mechanism to perform exact search in the swarm.

The API exposed by this type of service includes the following events:

Operations

FindNodes(byte[] requestID, byte[] searchData) → (byte[] requestID, Set<Host>) The FindNodes operation is the foundation to route information to a given node or set of nodes. It provides a (decentralised) functionality that allows an element of a swarm to collect (with assistance of other elements in the swarm) information about the closest nodes in the system that are logically closest to a given identifier.

2.5.9.3 Resource Storage Service

The Resource Storage service interface provides operations that can be leveraged by protocols implementing mechanisms for resource storage and location, which is a common usage of Distributed Hash Tables (or DHTs). By resource we mean any digital asset, which can include a file, some information, or other.

Many protocols that provide the Routing service, will also provide Resource Storage, however we consider that both services have significant differences regarding their operations and can be presented as two distinct services.

Operations

PutResource(byte[] key, byte[] data) The PutResource operation allows storing a new resource on the swarm system.

GetResource(byte[] requestID, byte[] key) → (byte[] requestID, Boolean found, byte[] key, byte[] data) This operation allows the retrieval of a resource, stored on the swarm system, by providing the respective key.

RemoveResource(byte[] key) The RemoveResource operation allows for the removal of a resource stored on the system, given its identifier.

Notifications

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

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

2.5.9.4 Dissemination

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

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

Operations

Disseminate(byte[] data) The Disseminate operation allows to delegate the dissemination of data to some dissemination decentralised service by an application (or other protocol). The guarantees of this communication primitive are protocol-dependent.

Notifications

DataReceived(byte[] data) A DataReceived notification can be triggered by protocols to applications or other protocols that consume information received from the Dissemination service. The notification is triggered whenever new data, disseminated by any node on the network, is received.

2.5.9.5 Interaction Model

The generic interfaces are special protocols (using the Babel terminology) that mediate the interaction between applications/protocols and specific decentralised protocols. Each interface exposes the set of operations and notifications discussed above.

While Babel, and the strategy currently put forward by the TaRDIS project and discussed in Deliverable D3.1, is to take advantage of a programing model based on state machines, where the interaction between components happen through asynchronous events, this approach is sometimes cumbersome for programmers, particularly when two components interact using a request-reply model, where component A requests something of component B that it requires to complete the task it is currently executing. In such a setting a blocking interface might make the code of component A simpler and much less prone to errors.

To overcome the limitation of this programming model, we have decided to experiment with supporting three distinct interaction models between components (in this case Babel protocols) using a single implementation, and dealing with this at the Generic Service Interface. The three interaction models we consider are the following.

- A fully asynchronous interaction based on event-driven mechanisms with requests and replies;

- A Promise-based interaction where requesters receive a promise (or future) and can execute other tasks until they block until the promise can be resolved (i.e., they can block their execution until the reply is received;

● A synchronous (blocking) interaction where the requester always waits until the operation is complete whenever they execute a request to another component.

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

Further laboratory experimentation with this has shown that there was some overhead when mediating the interaction between protocols using this mechanism, and as such we have left this methodology as a proof of concept to which we can return further ahead in the project execution to exploit for particular stacks of protocols in Babel that can be provided as a standalone black box component for programmers that provide a given abstraction and that they can use as a middleware.

However, we integrated the lessons learned from these efforts to materialise a common event-based API for protocols in Babel that provide the same service/functionality for a swarm application. We have dubbed this **Babel Protocol Commons**, that we now describe here as a way to complement the evolution of the tooling reported in this section.

2.5.9.6 Babel Protocol Commons (API description)

Babel protocol commons provide common events that serve as API and should be integrated by any protocol that offers an abstraction that can be mapped to either *membership*, *dissemination*, or *storage*. We have made a significant (but not yet complete) effort to ensure that the API defined in Babel Protocol Commons adhere to the TaRDIS API previously presented in Deliverable 3.1. This tool is still evolving to support other tools currently being developed within the context of WP6.

Membership Services

Requests:

GetNeighborsSampleRequest: Request to get a sample of neighbours (Host format) from the partial view up to a number (provided in the request).

Replies:

GetNeighborsSampleReply: It contains a set of peers (in Host format) and is generated in response to the previously described GetNeighborsSampleRequest.

Notifications:

NeighborUp: indicated the Host of a local neighbour that became available

NeighborDown: indicates the Host of a local neighbour that is no longer available

Dissemination Services

Requests:

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

Replies:

There are no Replies defined in the dissemination service.

Notifications:

BrodcastDelivery: Indicates that a broadcast message with a `payload` has been received (notice that the issuer of a BroadcastRequest will also receive an asynchronous notification with the contents that were broadcasted).

Storage Services

Requests:

CreateKeySpaceRequest: A request to create a key space in a data management protocol, with a given set of `properties`.

CreateCollectionRequest: Request to create a `collection` (akin to a table) in a specific `keySpace` in a storage protocol, with a given set of `properties`.

ExecuteRequest: Request to execute an operation on a specific `keySpace` and `collection`. The operation in question is specified through the abstract class `CommonOperation` which can be instantiated with a specific operation type (i.e., `SmartContractOperation`, `PayloadOperation`, etc.).

DeleteKeySpaceRequest: Request to delete a `keySpace`.

DeleteCollectionRequest: Request to delete a `collection` in a `keySpace` .

Replies:

CreateKeySpaceReply: Generated in response to a `CreateKeySpaceRequest`, with the status of the operation and an optional message.

CreateCollectionReply: Generated in response to a `CreateCollectionRequest`, with the status of the operation and an optional message.

ExecuteJSONReply: Generated in response to an `ExecuteRequest`, with the status of the operation and the response data (if any). This class parses the result as a JSON to be passed on to the application.

ExecutePayloadReply: Generated in response to an `ExecuteRequest`, with the status of the operation and the response data (if any). This class stores the result as a payload (a byte[]).

ExecuteSatusReply: Generated in response to an `ExecuteRequest`, with the status of the operation. This reply is meant for responses that don't contain any data.

DeleteKeySpaceReply: Generated in response to a `DeleteKeySpaceRequest`, with the status of the operation and an optional message.

DeleteCollectionReply: Generated in response to a `DeleteCollectionRequest`, with the status of the operation and an optional message.

NotSupportedReply: Generated in response to a request for an operation that is not implemented in the underneath storage protocol.

Notifications:

The nature of storage services being explored by TaRDIS at this point do not generate asynchronous notifications.

2.5.9.7 Distribution

Generic API for Decentralised Overlay & Communication Protocols

The original tool providing a generic API for decentralised membership and communication abstractions is open source and available at:

https://codelab.fct.unl.pt/di/research/tardis/wp6/overlayapi

Examples of its usage are also offered as open source software and available in the following TaRDIS repository:

https://codelab.fct.unl.pt/di/research/tardis/wp6/overlayapiexamples

Babel Protocol Commons

Babel Protocol Commons has two different variants that are similar except that the first was designed for the original Babel framework and the latter was adapted (due of necessity to deal with dependency management in maven) for the new Babel Swarm framework that has being developed in the context of TaRDIS (these are discussed in further detail ahead in Section 2.5.4 as stated previously). Both are provided as open source in repositories of TaRDIS and also as maven dependencies. Below we provide information on both variants for completeness.

Babel version

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

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>

<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-commons</artifactId>
    <version>[1.0.17,)</version>
</dependency>
```

Babel-Swarm version

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

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>

<dependency>
```

```
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-commons-j21</artifactId>
    <version>[1.0.16,)</version>
</dependency>
```

We should note that the name of these artefacts might change, as the technology being developed in TaRDIS further matures, it is our plan that the Babel Swarm variant (and all its support libraries, which is the case of Babel Protocol Commons, will become the defacto main distributed version of Babel.

2.5.10 T-WP6-02 An Epidemic and Scalable Global Membership Service

The Epidemic and Scalable Global Membership service is a tool provided in the form of a library, that implements a distributed and decentralised protocol to offer to swarm applications or protocols used in the implementation of a swarm application an eventually consistent global view of the system membership. This was previously reported as part of the output of WP6 in Deliverable 6.1. We provide a brief summary of this tool here for self-containment.

It should be noted that most of the membership abstractions currently being explored in the context of TaRDIS are based on keeping available, locally at each swarm element, information about a small subset of other participants in the swarm. This is an approach originally developed in the context of peer-to-peer systems, and its benefits is that it significantly reduces the overhead of maintaining membership information available to each node in a decentralised system up-to-date, ensuring availability and scalability, which are primary concerns in swarm systems.

However, examining the requirements of some use cases in TaRDIS (as detailed previously in Deliverable D2.2, and also revisited in this deliverable) we have identified the need for providing potentially transiently inaccurate full membership information. In fact, we make the observation that non-essential tasks could benefit from having an eventually correct notion of a global membership. This can be useful to collect statistics from the operation of the system, or to feed information for consoles that report the status of the system to human operators.

This however goes against the principles discussed above to ensure scalability and availability. To circumvent this limitation of global membership services for swarms we developed a new solution that, in addition to providing a global (i.e., complete) eventually correct membership view to each node, operates by taking advantage of a partial-view based membership abstraction, and a decentralised broadcast communication abstraction, which is a perfect illustration of the composability of components offered by the TaRDIS toolkit to build either other tools or to offer additional guarantees.

More interestingly, our design does not prescribe or depends on concrete membership and communication protocols, being sufficient that the membership service ensure global connectivity at the random graph denoted by the partial views maintained by each node in the swarm and that the broadcast protocol provides probabilistic atomic broadcast with a configurable high probability.

While the design of this solution and the way that we leverage on these (somewhat simpler) abstractions of membership and communication are discussed in detail in D6.1, the main intuition is that nodes in the swarm are part of the scalable membership service based on partial views, and rely on the epidemic broadcast to announce themselves in the network, ensuring that everyone node in the swarm become aware of their existence at the level of the Epidemic and Scalable Global Membership Service. Furthermore, when the direct neighbours of a swarm node become aware of the departure of or suspect of their failure

they can use the same principle to expedite the removal of that element from every other node global view.

2.5.10.1 Distribution

The Epidemic and Scalable Global Membership protocol is available as open source in the following TaRDIS repository:

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

The protocol is also available as a Babel java library that can be imported using maven adding the following directives to a pom.xml (we should note that due to lack of time this protocol is still only available for the original Babel, with a version adapted for Babel Swarm to be released at a future date):

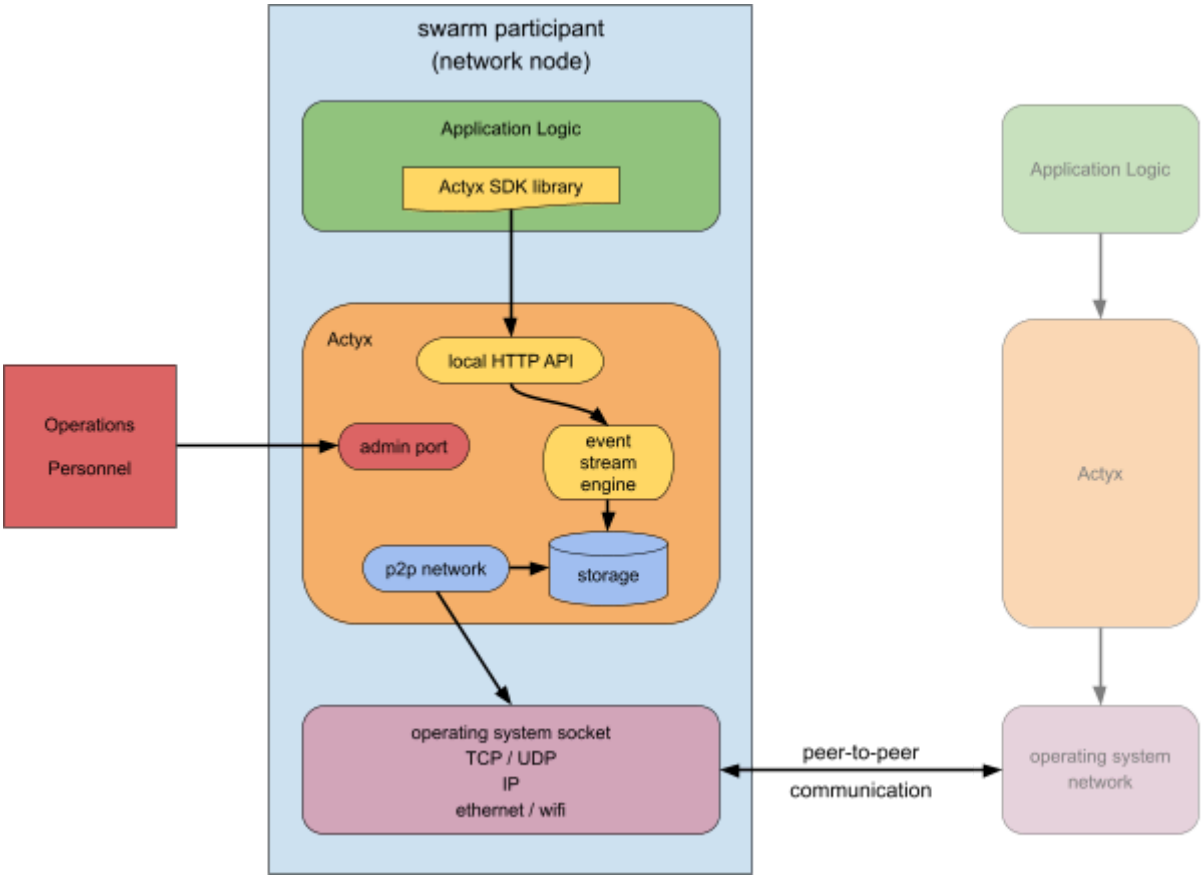
```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>

<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-epiglobalview</artifactId>
    <version>[1.0.0,)</version>
</dependency>
```

2.5.11 T-WP6-03 Actyx: Reliable event broadcast with configurable durability

Actyx is a middleware tool that will internally use the above tools for swarm communication and membership to provide even higher level services to TaRDIS applications, namely the reliable and durable dissemination of event streams within the swarm. This is significant because individual events are quite small and typically don't warrant the overhead of being individually treated (e.g., for being identifiable or localizable) in a swarm system. Therefore, Actyx partitions the emitted events into streams that are then the unit of dissemination, leading to significant benefits in compressed event storage size. Storage resource usage can be controlled via configurable per-stream data retention policies. Actyx also introduces an eventually consistent global order between events that allows the resolution of conflicts arising from concurrent swarm behaviour in a fashion that does not compromise on system availability or resilience.

2.5.11.1 Tool Architecture



Actyx tool architecture

Actyx runs on each device participating in the swarm as a separate operating system process, much like database management systems do. The TaRDIS application using Actyx runs in its own process, driven for example by the node.js runtime environment. The application makes use of Actyx facilities by sending requests and receiving responses across a WebSocket connection established via the local loopback network interface.

Internally, Actyx hosts the HTTP API applications connect to, which uses the custom event stream engine to respond to queries, implementing the Actyx Query Language (AQL). Events are persisted and retrieved via a local storage module whose contents are synchronised with other Actyx nodes in the swarm by the peer-to-peer network module. Various aspects of Actyx can be configured via the admin port that is accessed using the Actyx CLI or Node Manager applications; examples include managing cryptographic keys, data retention policies, network connectivity parameters, and also a wide range of diagnostic information to assess the health of this node in the context of the swarm.

The current Actyx production version uses libp2p to implement the peer-to-peer communication stack, which of course is based on the operating system's socket abstraction for TCP/IP and UDP/IP networking. The storage employs the SQLite embedded database and a custom data schema that allows fully content-addressed access to all information and offers persistent append-only data structures to the event stream engine.

In the second part of the TaRDIS project we aim to incorporate techniques from the other tools in WP6 to replace or enhance both the networking and storage parts of Actyx. In particular, we plan to use hierarchical overlay networks to make swarm connectivity

management more efficient and allow for partial replication of data based on local application needs.

2.5.11.2 Tool Dynamic Behaviour

The application can inject information into the swarm by means of emitting an event. This event is stored locally by the Actyx process and it is also communicated to the rest of the swarm via the protocols and services described in the sections above (in particular overlay networks and membership service). The local application on any swarm node can then query the locally stored events and thus receive the information.

2.5.11.3 Tool Persistence

Event data is stored in highly compressed binary format by Actyx on the file system provided by the host operating system, using the SQLite embedded database system.

2.5.11.4 Tool Operational details

Actyx can be installed by downloading the binary executable file or by building it using the Rust package manager:

```
cargo install ax
```

The Actyx process is created from the command line `ax run` while the rest of the `ax` subcommands offer management and introspection tools, like

```
ax nodes inspect localhost
```

to see the local Actyx process' list of peers, connections, network problems, etc.

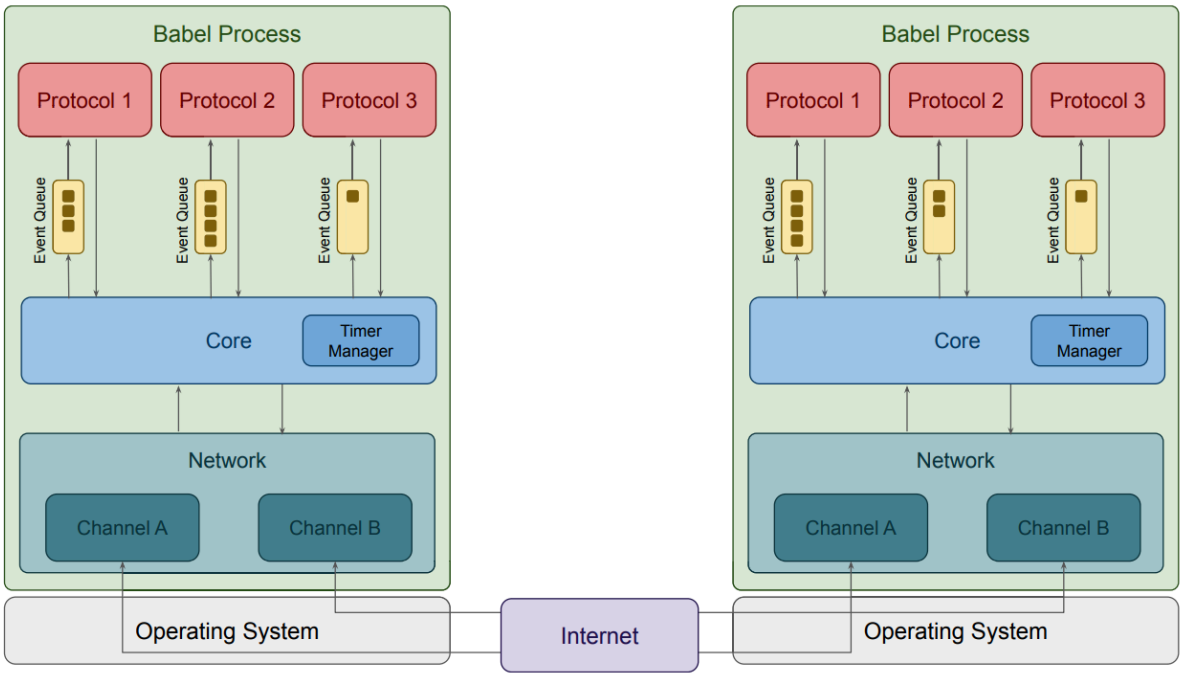
2.5.12 T-WP6-04 Babel (extended to support autonomous swarms)

Babel[32] is a framework developed at NOVA, prior to the start of TaRDIS, whose goal is both to simplify the development of distributed/decentralised protocols and systems, by providing a modular programming model where developers can combine different distributed protocols implementations to build their applications, and to provide an effective runtime that can support the execution and interaction (within a single process and across processes) of different protocols whose combination provides the fundamental support and abstractions to support the operation of the distributed application/system being developed.

While Babel was designed to support the needs of generic distributed systems, and it was internally used to successfully implement distributed storage systems, state machine replication systems, consensus primitives and several peer-to-peer protocols (as discussed previously) it lacked support for advanced features required to build autonomous and heterogeneous swarms. We have therefore evolved Babel to have a new set of features towards this end. In this section we start by discussing the architecture and abstractions provided by the original Babel framework, and we then discuss the new aspects in the new version of Babel that we have been developing to which we call Babel-Swarm.

[32] 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.

2.5.12.1 Tool Architecture



Babel tool architecture

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

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

Protocols:

Protocols are implemented by developers (i.e., the users of the Babel framework), and encode all the behaviour of the distributed system being designed. Each protocol is modelled as a state machine whose state evolved by the reception and processing of (external) events. For this purpose, each protocol contains an event queue from which events are retrieved. These events can be Timers, Channel Notifications from the network layer,

Network Messages originated from another process, or Intra-process events used by protocols to interact among each other within the same process.

Each protocol is exclusively assigned a dedicated thread, which handles received events in a serial fashion by executing their respective callbacks. In a single Babel process, any number of protocols may be executing simultaneously, allowing multiple protocols to cooperate (i.e., multi-threaded execution), while shielding developers from concurrency issues, as all communication between protocols is done via message passing.

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

Core:

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

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

In addition to mediating interaction between protocols (both inter and intra process), the core also keeps track of timers setup by protocols, and delivers an event to a protocol whenever a timer setup by the protocol is triggered.

Network Channels:

Babel employs an abstraction for networking which we name channels[33]. Channels abstract all the complexity of dealing with networking, and each one provides different behaviours and guarantees. Protocols interact with channels using simple primitives (openConnection, sendMessage, closeConnection), and receive events from channels whenever something relevant happens. These events are channelspecific and are handled by protocols just like any other type of event (i.e., by registering a callback for each relevant channel event).

For instance, the framework provides a simple TCPChannel which allows protocols to establish and accept TCP connections to/from other processes. This channel generates events whenever an outgoing connection is established, fails to be established, or is disconnected, and also whenever an incoming connection is established or disconnected. Other examples of provided channels include a channel with explicit and automatic acknowledgement messages, a channel that creates one connection for each protocol running in different processes, and a ServerChannel that does not establish connections,

[33] The reader should notice that these channels are different from the channels discussed in deliverable D3.1 regarding communication abstractions. These are low level channels that are responsible for handling point-to-point interactions across different processes of a distributed system whereas the channels proposed in D3.1 as a TaRDIS abstraction are higher level abstractions, including point-to-multipoint abstractions. The latter can be materialised through the adequate combination of protocols executing within Babel however.

only accepts them, and its corresponding counterpart, the ClientChannel. We also provide a TCP-based channel that implements the ϕ-accrual failure detector[34], which notifies protocols that registered a callback whenever another process is suspected.

2.5.12.2 API

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

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

Inter-protocol communication: Our framework supports multiple protocols executing concurrently in the same process. As such, we offer mechanisms for these protocols to interact with each other, allowing them to cooperate or delegate responsibilities between them. For instance, we could create a solution where a message dissemination protocol takes advantage of a peer sampling protocol to obtain samples of the system's membership. We could take this example further ahead and use that dissemination protocol to support for instance a data replication protocol. Relationships between protocols do not need to be strictly one-to-one. For instance, a membership protocol can be used simultaneously by different protocols doing different tasks for the application.

In babel we support two distinct inter-protocol communication events, for two distinct interaction models. **Request-Reply events**, where a protocol (or the application) can send an explicit request to a protocol requesting some service or information, and a reply can be sent back to the requester providing the requested information or some acknowledgement of the requested service. This model of interaction implies that the requester is aware of the other protocol existence at implementation time. The alternative is asynchronous **Notification** events, that empower a protocol to issue information to the stack without being aware of who is going to receive that information. Any protocol or application can register with Babel to receive different types of notifications. This interaction model assumes a more decoupled interaction between protocols.

Messages: Naturally, as a framework for distributed protocols, Babel also provides abstractions to allow the exchange of events (in the form of messages) across different processes of a system (typically between instances of the same protocol which ensures that the contents of the message can be interpreted by the receiver, but this not mandatory.

Messages are sent between processes through a channel (described above) that can provide different point-to-point guarantees, such as ordering of messages, automatic retransmissions, etc. Therefore for a protocol (or application layer) to send messages to another process it must have access to a channel, either created by it or shared among different protocols and/or the application layer. In babel, for a question of efficiency in communication across processes, we require the programmer to define serializers/deserializers using the Netty framework mechanisms. These must be registered with Babel (in particular within the Babeel channel being used) to ensure that the framework knows how to serialise and deserialize these events. Akin to all other events in the API of

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

babel, the Babel core ensures that a message received though some channel is put into the event queue of the appropriate protocol for processing by its private thread.

2.5.12.3 Babel-Swarm

While the abstractions provided by Babel are enough to build correct and performant distributed systems, through experience, and highly motivated by the needs and challenges being tackled by the TaRDIS project, we have identified a set of pressure points in the development of swarm applications using Babel.

As discussed above, to address this we have significantly evolved the Babel framework to provide both programming and runtime support for these issues.

The issues that we have identified are the following:

- Swarm system can be operated by different entities, for instance consider a scenario such as the one from EDP, where a regular user might want to install a device running a component of a decentralised energy market swarm at their home, the key challenge here is that it should not be expected that the user knows how to correctly configure the application. While in some scenarios it might be enough to ship the application with a static configuration, in some cases this might not be possible, if the configuration must be modified for some reason. Also, many decentralised systems require an introduction node (also known as a *contact*), a member of the swarm that assists a new element to join the swarm and start interacting with members of it. Due to this we have made efforts to support multiple forms of **self-configuration** in Babel.
- Still related with the configuration that governs the behaviour of individual swarm components, as the environment where the swarm operates, workloads change, or even as the system grows by having new element join, the initial configuration of the swarm members might become sub-optimal, leading to disruption in the quality of service being provided to users, inefficient resource usage (CPU, memory, network, or even carbon footprint), and in the worst possible scenario, to the full disruption of the swarm operation and its failure. This points towards the need to support some form of autonomic reconfiguration of the swarm, where members of the swarm are able to collect information about their operation (and potentially also from some other members of the swarm), their environment, and local information about performance, and based on this modify their behaviour. The simplest way to do this is by changing the runtime parameters of components in the swarm (e.g., a membership service based on peer sampling might need to maintain information about additional elements in the system if the total number of nodes increases). Unfortunately, the original version of Babel implicitly assumes that these configuration parameters are static throughout the lifecycle of the application. Due to this we have evolved Babel to provide support for live **reconfiguration** of Babel protocol parameters.
- Finally, practical swarm systems, particularly those that execute continuously or that control safety-critical systems, for instance a factory as in the Actxy use case, must deal with security concerns. This means that protocols executing within Babel, and the Babel runtime might be required to explicitly deal with this, which originally Babel did not provide any form of native support for. Challenges faced in this context include, but are not limited to, managing the identity and authentication of elements ini the swarm, ensuring that only authorised elements can join and interact with the swarm, which might be difficult if no centralised entity can be leveraged. Communication channels might need to be protected to ensure both that private information is not exposed to evesdroppers or that information in transit is not manipulated. Protocols might need to access and manipulate cryptographic material (such as long term asymmetric key pairs, or session keys) to support their operation.

These can be time consuming and error prone tasks and hence we have evolved Babeel to provide additional **security abstractions**.

The concrete mechanisms being provided by this new version of Babel that we call Babel-Swarm (with internal development identifier babel-sc-core) will be further detailed and specified in the upcoming D6.2 deliverable.

2.5.12.4 Distribution

The Babel framework is open source, and its (most recent) version can be found in:

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

Babel is distributed as a java library through maven also using the following configurations in the pom.xml file:

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>

<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-core</artifactId>
    <version>[1.0.5,)</version>
</dependency>
```

The new Babel-Swarm version is still being developed and evaluated. Its code is currently accessible at:

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

Similar to the original Babel, Babel-Swarm is distributed as a java library through maven, and can be accessed using the following configurations in the pom.xml file of a Java project:

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>

<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-sc-core</artifactId>
    <version>[0.5.8,)</version>
</dependency>
```

2.5.13 T-WP6-05 Arboreal: Cloud–Edge Data Management, Dynamic Replication

We expect some swarm systems to take advantage of external highly available and fault-tolerant storage abstractions. While in some cases the use of a cloud-based replicated

or geo-replicated storage service such as Cassadra[35], MongoDB[36], CosmosDB[37], or other will be sufficient, particularly if the operation of the swarm is not highly dependent on data stored and managed by these solutions, we also envision some applications for swarms where interaction with such solutions are in the critical path of the swarm operation. For instance, consider a federated learning service such as the one put forward by Telefonica. In this application we can (and probably should) delegate the application state to an external storage service, however this storage solution will be frequently accessed by participants during federated learning activities to both store their weights after an iteration, to collect the weights of several participants to aggregate them, and to access the weights of the next iteration. To provide such features, in TaRDIS we have developed a distributed data management system that extends from cloud-infrastructures across the edge continuum, and that as we detail ahead, combine a unique set of features that makes it an suitable and attractive solution for supporting large-scale swarm systems. Interestingly, and similarly to the intelligent swarm systems envisioned by the TaRDIS consortium, this solution, that we have named Arboreal, also features autonomous behaviours and managements, which we believe will contribute significantly for its success in the context of swarms. Arboreal has been accepted for publication and will be presented to the scientific community soon[38].

2.5.13.1 Motivation and Design Principles

The current deployment and operation model of Internet services consists in running the main application components in cloud infrastructures, accessing data that is managed by some (potentially geo-replicated) storage system. In this model, edge locations, that include cloud operators points-of-presence, ISP infrastructures and Internet exchange points, or small regional data centres, only execute simple tasks that access static or cached data. To fully realise the potential benefits of edge computing to provide adequate support for large-scale applications that are sensible to data access latency, with elements of those systems scattered across large geographical areas, it is necessary to adopt a different model, where edge nodes can process all clients requests. Doing this efficiently, requires unrestricted access to application data across different geographical locations where its components might be running.

To support this model, storage systems need to expand from executing only on data centres to edge locations, addressing challenges that depart from those addressed by traditional cloud storage systems:

Partial and dynamic replication. We expect applications with large user bases or large number of devices to take advantage of, similarly, a large number edge locations, one or more orders of magnitude above the typical number of data centres used by current cloud-centric applications. These edge locations feature computational resources that are less powerful and reliable than those found in core data centres. Regardless of executing on the cloud or in edge locations, application components should have full access to their data with the same guarantees.

This requires data storage solutions featuring fine-grained partial and dynamic replication, where the data objects that are replicated at each edge location change over time to reflect the access patterns of the operations received from clients or application components in

[35] Lakshman, Avinash, and Prashant Malik. "Cassandra: a decentralized structured storage system." *ACM SIGOPS operating systems review* 44.2 (2010): 35-40.
[36] Bradshaw, Shannon, Eoin Brazil, and Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media, 2019.
[37] Guay Paz, José Rolando, and José Rolando Guay Paz. "Introduction to azure cosmos db." *Microsoft Azure Cosmos DB Revealed: A Multi-Model Database Designed for the Cloud* (2018): 1-23.
[38] Large Scale Causal Data Replication for Stateful Edge Applications. Pedro Fouto, Nuno Preguiça, and João Leitão. Proceedings of 44th IEEE International Conference on Distributed Computing Systems, New Jersey, USA, July 2024 (to appear).

close vicinity to that location. This is challenging because the replication protocol will have to adapt itself to ensure that data updates are propagated to the correct locations, to avoid application components and clients to observe stale data for long periods, or for their operations to never become visible across the rest of the system.

Scalable consistency. To ensure that the application logic can remain mostly unchanged and independent of the location where it executes, some form of data consistency must be provided. While strong consistency allows simplifying the application logic, by avoiding data anomalies, such approach is not suitable for edge settings, as the latency incurred by the coordination of a large number of replicas would defeat the benefit of decentralising data replicas towards edge locations. Furthermore, the availability of the data storage system could become compromised whenever a replica becomes unavailable. For an edge environment, it is more suitable to rely on a weak consistency model, which sacrifices consistency for better availability and response times. To minimise anomalies exposed by eventual consistency models, many solutions opt to provide causal+ consistency[39] which provides the strongest consistency guarantees while allowing the system to remain available when some replicas are unreachable.

Causal+ consistency fundamentally captures the happens-before relationship, where the effects of a write operation can only be made visible in a replica after the effects of all writes that causally precede it have locally been applied. Concurrent writes however, can be made visible in any order, as long as all replicas eventually converge to the same state. The main challenge of causal+ consistency lies in tracking the happens-before relationship across all operations. A common approach is to rely on vector clocks[40], but this approach does not scale, as the size of the vector clocks grows linearly with the number of replicas, resulting in a significant metadata overhead. Other approaches, such as tree-based topologies avoid growing metadata costs by leveraging the tree structure to propagate operations in a strict order that respects the happens-before relationship, however correctness depends on the tree topology which can easily be disrupted by changes in the replica set, with a negative impact on the overall fault tolerance of the storage system. Finally, some solutions have opted to rely on a centralised component to simplify causality tracking, which leads to a central contention point that limits the benefits of using several edge locations.

We address the challenge of finding a scalable solution for tracking causal dependencies across write operations that can scale to hundreds of locations, in a context where data object replicas are dynamically managed; and in a way that can deal with frequent replica set changes and be fault-tolerant.

Client Mobility. Many applications that can benefit from the envisioned stateful edge applications have large numbers of users scattered throughout different locations, with the data being accessed depending on their location. Examples of such applications, in addition to decentralised federated learning using user-data and mobile clients, include collaborative applications (e.g., Google Docs), autonomous vehicles, smart-city applications, multiplayer mobile games, or stateful serverless computing. In such applications, where low response times are important, users might change location while using the applications. When doing so, it should be possible for a user to migrate from interacting with the data storage system replica deployed on an edge location to a closer one (that can provide faster response

[39] Lloyd, Wyatt, et al. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS." Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. 2011.
Almeida, Sérgio, João Leitão, and Luís Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication." Proceedings of the 8th ACM European Conference on Computer Systems. 2013.
P. Fouto, J. Leitão, and N. Preguiça. “Practical and Fast Causal Consistent Partial Geo-Replication.” Proceedings of the 17th IEEE International Symposium on Network Computing and Applications (NCA 2018). November 1-3, 2018. Cambridge, USA.
[40] Baquero, Carlos, and Nuno Preguiça. "Why Logical Clocks are Easy: Sometimes all you need is the right language." *Queue* 14.1 (2016): 53-69.

times). However, when this happens, the application state exposed to the user should not make any anomalies noticeable by application and users alike. This points towards the need of a distributed data storage solution able to handle mobile clients without sacrificing consistency guarantees.

2.5.13.2 Architecture and Design

Arboreal is a distributed data management system for the edge that, at its core, features a novel scalable replication protocol that provides causal+ consistency. Arboreal can scale to hundreds of edge locations, being able to adapt to changes in the membership and transparently deal with faults and network partitions. It relies on dynamic partial replication, allowing edge nodes to automatically adapt the set of data objects that they replicate locally to adjust to changes in the access patterns of application components interacting with that replica directly, while simultaneously allowing these *clients* to move across edge locations without sacrificing consistency guarantees. All this is achieved in a fully decentralised manner.

System Model: The design of Arboreal assumes a set of cloud data centres spread across different geographic regions in which a distributed NoSQL database is running. The deployment details of this database on the data centres (e.g., replication protocol, partitioning scheme) is orthogonal to the operation of Arboreal. We consider a set of edge locations with available computational resources. Arboreal extends the database running on the cloud data centres to these edge locations of their respective regions. For this, an instance of Arboreal is deployed in the cloud data centre of each region, and in each edge location. We assume that edge locations can be composed of one or multiple edge nodes (e.g., a small data centre), however Arboreal treats each edge location as a single (virtual) node (i.e., a logically single instance of Arboreal is deployed in each edge location) and does not make any assumptions about possible replication and data partitioning schemes that may be used inside each edge location, focusing instead on the replication of data across different edge locations.

To support a wide range of scenarios and applications, we assume that Arboreal can be dynamically deployed across new edge locations at any time, and that edge locations can fail (or Arboreal can be decommissioned in them) at any time. We further assume that the access pattern to data at each edge location can change over time.

Data Model: Arboreal provides a key-value store interface, similar to other highly available distributed databases, where each data object is identified by a unique key. Clients of Arboreal interact with it by issuing operations to read or write data objects without restrictions, and it is Arboreal's responsibility to ensure that: (1) data objects are transparently replicated to the edge locations where they are accessed; (2) client write operations are propagated to all locations that replicate the data object being modified (at that time); and (3) clients always observe a state that respects the causal order of operations. Arboreal makes no assumption about how the data is actually stored in each node. For convergence, Arboreal relies on a last-writer-wins policy, based on the operations timestamps and the IP address of the node that generated the operation.

A key novelty of Arboreal is the techniques used to overcome the inherent limitations of replication protocols providing causal+ consistency in a way that is suitable for a large-scale edge environment. To address this challenge, it is paramount to allow edge locations to synchronise (i.e., propagate operations and data objects) directly with each other, while simultaneously ensuring that metadata, used for both tracking and enforcing causality as well as to support (object-grained) dynamic replication, and communication overheads do not grow linearly with the number of edge locations in the system.

Replication Model: Hierarchical Approach. The design of Arboreal is based on a hierarchical approach, with an instance of Arboreal being deployed in each edge location. Edge locations are then organised in a tree structure rooted at their regional data centre, which we call the region control tree. The management of the control tree is done in a fully decentralised manner, and nodes are not aware of the entire membership. Instead, each node only communicates with its parent and children, and only tracks detailed information about its children, and a small amount of control information about its ancestors (i.e., the nodes in the path between itself and the root of the control tree). This allows latency-sensitive operations, such as recovering from failures, client mobility, or creating new replicas of data objects to be performed in localised fashion.

To establish the control tree, when an instance of Arboreal is deployed on an edge location, it uses a heuristic to choose the most appropriate existing node in the control tree of its region to connect to. Note that the goal of Arboreal is to replicate data for applications running on edge locations, and is not an orchestrator that decides where and when to deploy the application components. As such, to allow Arboreal to adapt to different edge scenarios, both the heuristic used to define the control tree and the information used by the heuristic are configurable by the application developer.

As geographic locality is a key aspect of edge computing, our implementation of Arboreal uses the geographic distance between edge locations as the main metric to form the control tree. However, depending on the application, different metrics can be employed, such as the latency between edge nodes, client locations, or event predicting the future demand for the application. In Section 4 we detail how the control tree is formed in our implementation.

Enforcing causality: Causal dissemination. We leverage the hierarchical topology of Arboreal to enforce causality. An interesting property of connecting edge locations in a tree structure is that we can provide causal consistency without the need for any metadata[41]. For this, nodes form the control tree by connecting to their parent and children using FIFO channels. When a node receives a write operation from a channel (i.e., from the parent or a child), it atomically executes the operation locally and puts it on the outgoing queue of every other channel. Additionally, local operations are atomically added to the outgoing queues of all channels. The result of this is that operations are always enqueued (and thus, executed) after all their causal dependencies. This technique, however, only works if clients always issue operations to the same node and if the control tree is static (i,e., nodes do not fail or join the system), which are unrealistic assumptions for edge environments where edge nodes may fail or be network partitioned, and mobile clients are frequent.

Timestamping: To overcome these limitations, Arboreal employs Hybrid Logical Clocks (HLCs)[42] to enforce causal consistency when the causal dissemination mechanism is not enough. An HLC combines physical time, which allows time to advance monotonically in each node, with logical clocks, which allow the capture of causal relationships between operations even in the presence of physical clock anomalies. Whenever a client's write operation is received by a node, it is tagged with a timestamp computed by that node's local HLC. This timestamp is propagated along with the operation through the tree, and is stored along with the object data in each node. Additionally, leveraging on HLCs, Arboreal employs the notion of Branch Stable Time (BST). A BST is a timestamp that is computed individually by

[41] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. 2017. Saturn: A distributed metadata service for causal consistency. In Proceedings of the Twelfth European Conference on Computer Systems. 111–126.
Albert van der Linde, Pedro Fouto, João Leitão, and Nuno Preguiça. 2020. The intrinsic cost of causal consistency. In Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data. 1–6.

[42] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical physical clocks. In Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18. Springer, 17–32.

each node as the minimum between the node's current HLC time and the BST of each of its children. The BST of a node captures the fact that no operation with a lower timestamp will be generated by any node in its branch (we define a node's branch as the node itself, and all its descendants in the control tree). For nodes without children, the BST is simply its own HLC time. Each node periodically propagates its BST to its parent (so that it can update its own BST), and to its children. When propagating to its children, a node also includes the BST of all its ancestors. As a result of this process, each node keeps track of the BST of its children, and the BST of all its ancestors. This information is crucial to allow Arboreal to provide causal consistency when recovering from failures and for supporting mobile clients.

Dynamic and Partial Data Replication. An important aspect of edge computing is that it is not reasonable to expect that all edge locations will have resources to replicate the entire dataset of an application. As such, partial replication becomes a key aspect of Arboreal. Furthermore, in order to support a wide range of applications, Arboreal must be able to adapt not only to changes in the access patterns of clients but also to mobile clients that can change the edge location to which they are connected at any time. Therefore, unlike cloud-based data management systems which typically use static data partitions, Arboreal needs to allow edge nodes to dynamically change the set of data objects they replicate at any time with fine granularity. Unfortunately, keeping track of which nodes replicate which data objects across a large scale system can be costly and require a large amount of metadata to be propagated, particularly if both the set of nodes and the set of data objects are dynamic. To address this challenge we rely on the hierarchical topology.

In Arboreal, each data object is replicated individually to a subset of edge nodes. We do so in a way that any node always contains the data objects that its children replicate. This results in the formation of a subtree of the control tree for each data object, which we call the object's replication tree. When a client requests a data object that is not replicated in the edge node it is connected to, the edge node sends a request to its parent node asking it to be added to the object's replication tree. If the parent node is part of it, it will send the current version of the object to the child node and keep track that this child node is now replicating that object. If the parent node is not part of the replication tree, it will send a request to its parent node, and so on, until the request reaches a node that replicates the object (or the cloud data centre, which replicates all objects). This node will then send the object to its child node, which will do the same until the object reaches the node that originally requested it. At the end of this process, every replica between the node that requested the object and the node that responded to the request will have been added to the requested object's replication tree.

As the number of data objects in the system can be very large, and the amount of storage available in edge nodes is limited, Arboreal needs to be able to remove data objects from edge locations that are no longer being accessed. To do so, each node keeps track of the last time each data object was accessed by a client, and periodically runs a garbage collection process that removes the data objects that have not been accessed for a configurable amount of time. When removing a data object, a node informs its parent that it is no longer replicating that object. Note that nodes can only garbage collect objects that are not replicated to any of its children, otherwise the replication tree would become broken.

Data Persistence. Arboreal provides a mechanism allowing applications using it to specify the persistence level of write operations. Effectively, this allows applications to specify how many nodes upstream in the replication tree a write operation must reach before it is considered to be persisted, and the operation can terminate. This mechanism is particularly useful for applications that require data persistence guarantees on more unstable edge locations.

Fault Handling and Recovery. Due to its decentralised nature, node failures in Arboreal must be detected and recovered locally, without any centralised coordination. The main challenges in this context are: (1) rebuilding the control tree after node failures, which implicitly rebuilds all inscribed replication trees; (2) ensuring no consistency violations occur during the rebuilding process; and (3) ensuring data persistence during failures and reconfiguration.

Additional information on the implementation of Arboreal can be found in Deliverable 6.1.

2.5.13.3 API

Arboreal exposes a simple API that, while not entirely similar to the generic API for data management abstractions presented in Deliverable 3.1, can easily be adapted through using a wrapper. The API, that we present below, was devised to simplify the use of the solution in a mostly application and setting agnostic way.

`Read(partition, key) -> ReadResponse(data, lClock):` This operation allows us to obtain the value of a given data object identified by a unique key (belonging to a given partition). The response, in case of success, is composed by the value of that data object (referred to as data above) and a hybrid logical clock (labelled LClock above).

`Write(partition, key, value, persistenceLevel) -> WriteResponse(lClock):` This operation allows modifying the value associated with a data object identified by a key on a partition to a new value. The argument in the operation persistenceLevel is optional, and it specifies the minimum number of replicas, up to the central root node of Arboreal, that must execute the operation locally before a response can be sent back to the issuer. This response carries a hybrid logical clock that forces the issuer of the operation to move forward to that logical point in time in its execution (which can affect future operations, namely when migrating to a new replica).

`Migrate(lClock, path-to-root) -> MigrateResponse(path-to-root):` This operation can be issued by a client of Arboreal to switch the replica to which it is connected, which is relevant for instance for mobile clients. This operation is essential to ensure that after a migration, the data exposed to the client is consistent with its local causal history (i.e., that we enforce causal+ consistency after migrations). Upon requesting a migration the client contacts the new replica to where it wants to migrate and provides his local hybrid logical clock and the current path between his previous local replica and the regional data centre. When the client receives a reply it can start interacting with the replica and access or write objects. The reply also contains the path between the new replica and the data centre.

`FetchPartition(partition) -> FetchPartitionResponse():` This operation is an utility that allows a client to speed up replication of data for the replica to which it is connected when it is aware that its access pattern will manipulate all (or most) data objects in a partition. The only argument is the data partition that will be accessed to the client. The response happens before replication takes place and has no content other than conforming to the client that his operation was successfully received and will be processed.

In these operations lClock is always a Hybrid Logical Clock composed of two integers.

As it should be clear to the reader, data objects, identified by unique keys, are organised in partitions. This is a common technique in many cloud-based storage systems.

2.5.13.4 Distribution

Arboreal is provided as open source software, and the code can be found at the following TaRDIS repository:

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

Arboreal should be deployed at all locations where it will be executed by a human operator with direct access to the infrastructures where it is supposed to be running (both cloud and edge locations). The repository contains a script to assist in the deployment of an instance of Arboreal, and links to external repositories with client code[43] and examples used in our experimental work[44].

2.5.14 T-WP6-06 PotionDB: Strong Eventual Consistency under Partial Replication

As discussed in the context of the Arboreal data management solution, some swarm system design and implementation can greatly benefit from external storage management systems, that can simplify the access and manipulation of data to different swarm components, while at the same time contributing for the durability of application state and integration of other components into the system, such as business intelligence analytics, using more classical tools. While Arboreal deals with making application data to be available and mutable for edge locations in an autonomous way, Arboreal still prescribes the existence of some cloud-based system that it integrates with.

We now present the efforts of the TaRDIS consortium in one such solution, that can be (eventually) combined with Arboreal. The emphasis of this solution is to provide strong eventual consistency under partial replication across different (distant) geographical locations. The motivation for this is that as the number of data centres increases, so does the replication cost of full replication, which makes partial replication an attractive approach. Our solution is named PotionDB, a novel geo-distributed, partially replicated key-value store. PotionDB transaction management and replication algorithms provide transactional causal consistency for transactions that access local and remote objects. To enable the support of recurrent queries over geo-partitioned data, PotionDB provides efficient materialised views that allow these queries to be processed locally at all replicas.

2.5.14.1 Motivation and Design Principles

PotionDB is a distributed database designed for supporting global services deployed across multiple data centres. In these settings, (some) data items are only needed at some geographic locations. This makes that most data is tied to some geographic location, one can expect that the majority of accesses occur on that location. PotionDB adopts partial geo-replication, with data items being replicated only at some locations. This allows PotionDB to reduce replication cost when compared to solutions featuring full replication, saving on both storage, processing, and networking costs.

While naturally many application operations access data objects directly (using read/write operations) we also consider operations that might require data that results from an aggregation (e.g., max, min, avg, top-k) For supporting the latter, PotionDB provides materialised views that are the result of an aggregation over geo-partitioned data and provides algorithms to efficiently maintain these views consistent across multiple locations.

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 a view, v.

[43] https://codelab.fct.unl.pt/di/research/tardis/wp6/public/arboreal-others/arboreal-client
[44] https://codelab.fct.unl.pt/di/research/tardis/wp6/public/arboreal-others/arboreal-experiments

The value of a view v is defined as a function over the values of other object(s) 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.

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[45], guaranteeing that object replicas converge to a single state in the presence of concurrent updates.

Interface. PotionDB offers a key-value transactional interface. Which translates for access being executed over objects in the database being executed in the context of a transaction.

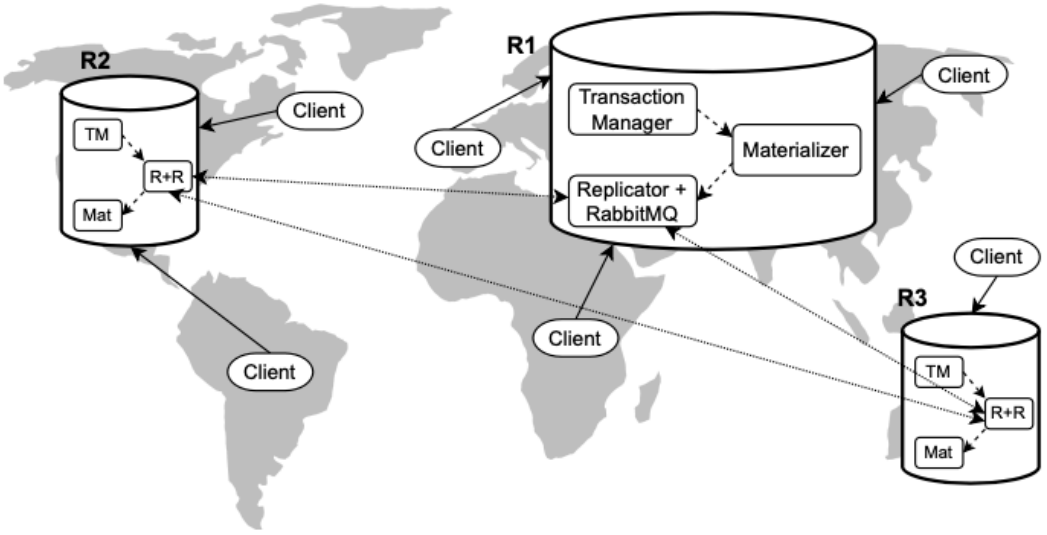
The create view receives a view specification defined in a language based on SQL (we provide an example further ahead). Given a view definition, PotionDB automatically infers the objects to be used to store the view and the view updates required to incrementally maintain the materialised view whenever a relevant base object is updated. The data in a materialised 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[46]. Intuitively, in TCC different replicas may execute transactions in different orders. A transaction accesses a causally-consistent database snapshot taken in the replica where the transaction is executed at the time the transaction starts. As in snapshot isolation, the snapshot reflects all updates of a transaction or none. Moreover, if a transaction t is included in the snapshot, all transactions that happened-before t are also included in the snapshot. Unlike snapshot isolation, and similarly to parallel snapshot isolation, it is possible for two concurrent transactions to modify the same object, with updates being merged using CRDT rules.

[45] Shapiro, Marc, et al. "Conflict-free replicated data types." Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13. Springer Berlin Heidelberg, 2011.

[46] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhong- miao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong semantics meets high availability and low latency. In 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). IEEE, 405–414.
Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. 2020. Transactional Causal Consistency for Serverless Computing. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (Portland, OR, USA) (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 83–97. https://doi.org/10.1145/3318464.3389710

2.5.14.2 Architecture and Design



PotionDB tool architecture

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

Clients communicate with the nearest PotionDB replica to ensure low latency. A client's transactions are locally executed in that PotionDB's replica. Updates are propagated asynchronously to other locations. Although we assume it to be uncommon, if a client's transaction accesses objects not locally available, locations where those objects are replicated are contacted and involved in the transaction execution.

The internal architecture of each PotionDB server is inspired by the Cure system and is split into three main components: (i) the Transaction Manager coordinates transaction execution, implementing a transactional protocol and enforcing the consistency of reads and updates; (ii) the Materializer stores the objects on their latest version, alongside the necessary data to generate previous versions when necessary. Garbage collection ensures data related with versions that are too old is eventually discarded; and (iii) the Replicator ensures that committed transactions are replicated asynchronously to other PotionDB instances. It is also responsible for receiving remote transactions and forwarding them to the Transaction Manager for local execution.

Data Object Representation

PotionDB stores two types of CRDTs: common CRDTs and non-uniform CRDTs (NuCRDTs)[47].

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 applied in an order that respects

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

causality. This is the case in PotionDB, as a valid transaction serialisation must respect the happens-before relation.

PotionDB supports the following CRDTs: last-writer-wins register, for storing opaque values; add-wins set, for keeping a set where adding an elements wins over a concurrent removal of that element; add-wins map, for maintaining a map of values; counter, for maintaining a number that accepts concurrent increment and decrement operations; average, for maintaining the average of values added to this object.

NuCRDTs. Non-uniform CRDTs 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 if, for each possible read operation, the result is equivalent when executed on either state.

Unlike normal CRDTs, in NuCRDTs, during the replication process, it is only necessary to propagate updates that may have an effect on the observable state. For example, consider a maximum object with 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 for a given NuCRDT object need to be replicated and applied everywhere. In PotionDB, we support the following NuCRDTs, previously proposed by Cabrita et. al.: maximum and minimum objects, for storing the maximum or minimum of the objects added to the object; top-K, for storing the K entries with largest values; top-K counter for storing the K map 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.

Sharding

PotionDB adopts a sharded model, where objects replicated in a location are split into multiple shards. For durability, each shard could be replicated in multiple servers using some replication protocol[48]. In each server, a shard has a dedicated thread, adopting an approach used in other database systems, such as H-store[49]. This avoids using locks when accessing objects, simplifying the implementation and avoiding issues such as lock contention.

Metadata

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

A transaction t has an associated read vector clock, t.rc, that represent the snapshot to be read by the transaction. On commit, a transaction is assigned a commit clock, t.ct, consisting in a pair (timestamp, location identifier). A transaction with commit clock (n, r_i) is in snapshot vc_G only if $vc_G[r_i] \geq n$.

[48] Pedro Fouto, Pedro Ákos Costa, Nuno Preguiça, and João Leitão. Babel: A Framework for Developing Performant and Dependable Distributed Protocols. Proceedings of the 41st International Symposium on Reliable Distributed Systems (SRDS 2022), September 19-22, Vienna, Austria, 2022.

[49] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2018. The end of an architectural era: It's time for a complete rewrite. In Making Databases 1505 Work: the Pragmatic Wisdom of Michael Stonebraker. 463–489.

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

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

Transaction Processing

A client executes a transaction by interactively contacting a PotionDB server. The execution of a transaction is coordinated by the Transaction Manager (TM). When the TM receives a begin operation, it decides the snapshot the transaction will access, i.e., the latest snapshot available at the current location, which is represented by the global vector clock vc_G.

When receiving an update operation, the TM asks the Materializer to execute the update in a private copy of the object for the transaction snapshot. When receiving a read operation, the TM asks the Materializer to execute the read operation in the private copy of the object - if no update has been executed before in the object, a shared copy with the version of the transaction snapshot is used.

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

Reads on NuCRDTs. As mentioned previously, due to the way Nu-CRDTs work, they may temporarily expose incorrect results when an operation modifies the set of relevant operations. PotionDB is able to detect this situation, which is expected to be rare. Applications may select two behaviours 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. The second option is to strictly enforce TCC. In this case, the read blocks until the replica gathers information that no relevant operation is missing. This requires contacting all replicas that maintain objects relevant to the operation.

Triggers. PotionDB has an after update trigger mechanism that can be associated with objects in a given bucket or container. When a transaction is executed at the initial replica, after an update, the code of the trigger will run and it may issue reads and updates to the same or to a different object. These updates are considered part of the transaction, being committed and propagated to other replicas together with the transaction. While triggers can be used directly by applications, their primary purpose is to support incremental view maintenance.

Replication

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

When a transaction commits at a shard, that shard sends its part of the transaction to the replicator. Given how transactions are committed at a shard, it is guaranteed a shard sends the transactions ordered by commit timestamp. If the shard processes no transaction for

some time, it notifies the replicator that there are no parts of transactions for the shard until the current timestamp, obtained from the shards' HLC.

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

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

Support for NuCRDTs. NuCRDTs use a non-uniform replication approach, in which some updates might not need to be propagated. 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 o may make a previous operation op relevant, requiring this previous operation to be propagated to other replicas (e.g., remove(n) operation in the maximum NuCRDTs makes the insertion of the second maximum element relevant). There are two cases to be considered. The first case is when operation o is executed initially. Then the effect of o will include also the effect of op . Thus, as the combined effects are propagated and applied in the context of the same transaction, no anomaly is generated. The second case happens when the now relevant op operation was executed in another replica. This is handled by the 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 o would generate the effects of op . These extra effects are remotely propagated along with the information that a given operation has been executed in the current location.

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 object used for storing the view's data depends on the type of query being performed. 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 view will be stored in a map CRDT, where the full materialised view is maintained. 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, such as count or average), the top-K counter is used. In any case, the elements of

the view are maps with multiple columns, one for each of the attributes specified in the select of the view definition.

For a given view definition, one or multiple objects are defined - in the language we defined for specifying views, a view that includes variables will lead to multiple objects, one for each possible value of the variables.

Generated triggers. After determining the objects used to represent the view, PotionDB generates triggers to update its contents, as base objects are created, updated or deleted. The trigger will be set for the container (or buckets) specified in the FROM clause of the view definition. If a where clause is included in the view definition, 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.

2.5.14.3 API

```
begin(clk) → txId

commit(txId) → clk

rollback(txId) → ok

get(txId, id) → value

read(txId, id, op) → value

upsert(txId, id, op) → ok
```

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. PotionDB also supports one-shot transactions, `oneShotTx(clk, (id, op)+)`, that includes 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, i.e., any object type can be stored in any bucket, we expect that applications store objects of the same type in each bucket. A document or a table row can be stored in PotionDB as a map CRDT, with each element of the map having its own type.

Funded by the European Union

```
CREATE VIEW (MonthlyTopSales, views) WITH
YEAR = ANY (SELECT DISTINCT SaleDate.Year FROM sales),
MONTH = ANY (SELECT DISTINCT SaleDate.Month FROM sales)
     AS
SELECT ProdName, SaleDate.Month, SaleDate.Year,
     SUM(Price - Cost) AS SumProfit, COUNT(*) AS NumSales
FROM Sales
WHERE SaleDate.Month = [MONTH] AND SaleDate.Year = [YEAR]
GROUP BY ProdName
ORDER BY SumProfit DESC, NumSales DESC
LIMIT 10
```

As stated before views are created using a language based in SQL. The Figure above provides a simple example of this. CREATE VIEW specifies the key prefix and bucket of the materialised 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. For instance we could use variables such as YEAR and MONTH to lead the system to maintain for each pair (year, month), the top 10. See the figure above.

2.5.14.4 Distribution

PotionDB is offered as open source software and can be obtained from the following public TaRDIS repository:

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

PotionDB should be deployed at all locations where it will be executed by a human operator with direct access to those infrastructures. The repository contains a README that provides indications on how to run the system.

2.5.15 T-WP6-07 Integration of Storage Solutions into the TaRDIS Ecosystem

Integrating third-party storage solutions into a system poses a range of challenges, emphasising the intricacies of aligning external tools with specific project requirements. However this is essential, on one hand to allow systems and applications developed under the TaRDIS approach to easily take advantage of well known existing abstractions, that have been proved by extensive usage by the community, and on the other hand to simplify application developed within TaRDIS, and using our toolbox, to more easily interface with legacy systems, for instance, by being able to consume data stored in well known storage solutions such as Casandra or even blockchains. The additional benefit that can be reaped from this integration is that we provide an unified API to interact with some of these systems, which contributed to modular designs, and provides the freedom to developers to try to use different systems to support their applications, and to swap these when the application requires a solution with somewhat different properties.

As previously reported in D6.1, we have achieved this by creating a set of *adapters*, implemented in the Babel framework (previously presented in this document) and that can easily be adapted to the new version of Babel currently under development (Babel-Swarm) to

integrate different storage solutions into the TaRDIS Ecosystem. To enable this, we created a set of abstractions (derived from the APIs presented in Deliverable 3.1).

These adapters essentially encapsulate within a Babel protocol, and expose an event-driven API, the client logic to interact with several storage solutions. Due to the different nature of different storage solutions (i.e., inserting a row in Cassandra presents different semantics from executing a transaction on the HyperLedger Fabric blockchain solution), we offer some flexibility to the developer by allowing different types of operations payloads (materialised as *abstract classes)* when executing an operation.

2.5.15.1 Current Integrations

So far, as part of the TaRDIS efforts we have integrated the following systems into our API through the aforementioned adaptors. We plan in the future to extend this set of systems if required explicitly by TaRDIS use case during their development, or if there is specific interest shown by the community:

- **Arboreal:**
  This adapter implements the Arboreal client-component. Arboreal is a generic solution for data management in the cloud-edge continuum that was developed as part of the TaRDIS activities, and previously mentioned in the report on Section 2.5.13.
- **Hyperledger Fabric (Blockchain system):**
  This adapter implements the Hyperledger Fabric client-gateway. In more detail, this adapter is responsible for invoking transactions on smart contracts deployed in the fabric blockchain network. Hyperledger Fabric is a platform for distributed ledger solutions that presents a highly modular architecture allowing confidentiality, resiliency, flexibility, and scalability[50].
- **C3:**
  This adapter implements the C3 client-library logic. C3[51] is designed to extend existing cloud-based storage systems by integrating a replication schema that can enforce causal+ consistency. At the moment this adapter implements C3 integration with Cassandra.
- **Cassandra:**
  This adapter implements Cassandra client-side logic. Cassandra is a highly performant distributed database, providing high availability and being proven across multiple systems across the world to be highly fault-tolerance.
- **Engage:**
  This adapter implements Engage client-side. Engage[52] is a storage system that offers efficient support for user/application-defined session guarantees in a partially replicated edge setting.

2.5.15.2 API

As discussed above the API offered by this tool (and the set of adapters reported above) is based on Babel events, making the interaction with the systems discussed above (and that

[50] This adapter was included into this tool thinking about the potential need of the EDP use case for a blockchain to register transactions among prosumers.
[51] P. Fouto, J. Leitão and N. Preguiça, "Practical and Fast Causal Consistent Partial Geo-Replication," 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), Cambridge, MA, USA, 2018, pp. 1-10, doi: 10.1109/NCA.2018.8548067.
[52] M. Belém, P. Fouto, T. Lykhenko, J. Leitão, N. Preguiça and L. Rodrigues, "Engage: Session Guarantees for the Edge," 2022 International Conference on Computer Communications and Networks (ICCCN), Honolulu, HI, USA, 2022, pp. 1-10, doi: 10.1109/ICCCN54977.2022.9868846.

are mostly external to TaRDIS) to be unified with other protocol interactions within Babel (and Babel Swarm). Therefore, and as previously done, the API can be described in terms of a set of (Babel) events. In this particular case, these events are restricted to Requests and Replies (listed below for self-containment of presentation in this document). Notice that if in the future we integrate a storage solution that can asynchronously interact with the application to notify of changes in the application data managed by it, this API will have to be extended to also include (asynchronous) notifications.

Requests

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

Replies

- *CreateReply*: Generated in response to a `CreateDataspaceRequest` or `CreateDataspaceRequest`, in reports on the status of the operation (i.e., success or failure).
- *ExecuteReply*: Generated in response to an `ExecuteRequest`, it reports on the status of the operation and the requested data (e.g., a `payload`) in case of success.
- *DeleteReply*: Generated in response to an `ExecuteRequest`, `DeleteKeySpace` or `DeleteDataSpace`, reporting on the status of the operation.
- *NotSupportedReply*: Generated in response to a request for an operation that is not implemented or supported by the underlying data management system (for instance a BlockChainOperation being issued to a system that is not a blockchain system).

2.5.15.3 Distribution

This tool is provided as open source, and the code is publicly available at the following TaRDIS repository:

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

Moreover, and similarly to the other tools in our toolbox that are part of the general Babel development framework, these adaptors can be easily integrated into an application by adding the following dependency to a project pom.xml used by maven (which is served from one of our repositories:

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>

<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-storage-adapters</artifactId>
    <version>[0.0.1,)</version>
</dependency>
```

2.5.16 T-WP6-08 Distributed Management of Configuration based on Namespaces

Swarm systems are naturally complex, this derives not only from their (mostly) decentralised nature, but also due to their heterogeneous nature but also due to potentially different administrative domains across different components (i.e., different participants in the swarm might be managed by different entities or users). Moreover, long-lived swarm systems will have to be adapted at runtime to avoid operating, for instance, with access to the incorrect amount of computational resources within a given computational infrastructure.

One approach to simplify such runtime management is to take advantage of containerization technology such as the one provided by solutions like Docker. This has the benefit of simplifying the deployment of swarm components, and also requiring less technical knowledge of users or administrator carrying out such deployment (remember that one of the goals of TaRDIS is no only to develop new swarm-enabling technology, but also to make it readily available, among others, by lowering the level of technical expertise required to developed and manage such systems).

In addition to taking advantage of containerization, we propose a mechanism to manage such swarm components based on hierarchical namespaces, which allow operators of the system to manage sets of swarm elements based on their assigned namespace. In the future, as we further explore self-management of systems, we can also take advantage of this to simplify the autonomic management of swarms by issuing runtime adaptations that affect only certain segments of the this hierarchical namespace, enabling a natural way to group elements of the swarm that share communalities or that are simply operated by the same entity. Configuration based on hierarchical namespaces can also help with naming collisions, which in such complex systems can occur frequently.

2.5.16.1 Architecture and Design

Our approach for Distributed Management of Configuration based on Namespaces is designed with scalability and fault tolerance in mind. Every part of the system is designed as a separate service designed for single entity purpose and using open source tools.

Separation of services allow this tool to scale only individual parts of the system that are required to be scaled, and not the entire system. Collaboration between these services allows better separation of concerns.

The system is composed of multiple services that manage: (i) Infrastructure, (ii) Namespaces, (iii) Schemas, (iv) Configuration, (v) Users etc. All services are implemented using go programming language and gRPC as a service communication layer.

2.5.16.2 API

Our tool feature the rich API presented below:

`createNS(cId, labels, resList) → nsaid`

`deleteNS(id) → status`

`readNS(id) → nsDetails`

`createChildNS(nsId, labels, resList) → nsId`

`createConfigSchema(labels, version, schema) → sId`

`deleteConfigSchema(sId) → sId`

`getConfigSchema(sId) → schema`

`getConfigSchemaTimeline(sId) → timeline`

`createConfig(labels, configList) → status`

`dissiminateConfig(labels, configList, percentage) → status`

`getConfigTimeline(config) → timeline`

Physical infrastructure can be splitted into multiple logically grouped entities using namespaces. Namespaces allow multi-tenancy meaning multiple applications can coexist on the same infrastructure, allowing better resource utilisation, but also prevent naming collision and easier configuration and reconfiguration management. In order to create a namespace, the user first must have setted up physical infrastructure and its unique identifier. To initiate creation of the namespace, the user calls `createNS(cId, labels, resList)`.

The user provides the infrastructure id, a set of labels, and a list of key-value pairs, that describes namespaces (e.g., name, location, purpose etc.), and how much computational resources this namespace is going to have. Notice, that number of resources must be less than or equal to the amount of resources of the parent, which in this case is physical infrastructure. We do not want to give more resources than are physically available within the runtime infrastructure.

When a namespace is created, the user will get back a namespace id. Existing namespaces can be deleted using `deleteNS(id)`, which will delete all namespace data. Users can also get namespace information using `readNS(id)`, or update information about namespace using the same `call for creation`. With this call, users can also require more resources for namespace from the physical infrastructure, and/or parent namespace.

Namespaces allow fine grain control over resources using parent-child relationships. Every namespace can further give its resources to its child namespaces created using `createChildNS(nsId, labels, resList)`. Notice that similar to creation of the namespaces, child namespace obey the same rule when requesting resources. Users can create a child namespace requesting some resources from its parent. The child namespace will be created only if the parent namespace has resources to give. When namespace structure is set up, resources are divided in the best way for organisation use cases, how users can send configurations for applications running inside them.

Child namespaces can be updated, deleted or retrieved with the same calls described earlier. namespaces To prevent sending various configuration formats and styles, users can create a schema, by which all configurations in that namespace will work. Using `createConfigSchema(labels, schema)`, users can create schema for specific purpose by providing labels (kay-value pairs to best describe schema), and also schema details in *YAML* format. This call servers also as schema update, but users must provide a new version! Here users can specify the structure of the configurations, values and types for every individual element, but also versions.

Schema versioning is important because users can go back in time using `getConfigSchemaTimeline(sId)`, and get schema changes over time and versions, to troubleshoot potential problems faster, or just to see how schema evolved over time. Schemas can be deleted using `deleteConfigSchema(sId)`, or users can get schema details using `getConfigSchema(sId).After setting up schemas, users can validate configurations, and propagate configurations to desired nodes.`

Using `createConfig(labels, configList)` users can deliver configurations to desired locations: (i) to the system and configure the system itself, or (ii) applications configuring applications. To do that, users have two options: (i) standalone configuration (set of key-value pairs), and (2) configuration group (a set of stand-alone configurations). With this we can configure a single element (e.g application, database, node etc.), or group of connected elements at the same time (multiple applications, application and database, application and cache etc.). Configurations are delivered to nodes in event-driven manner, and for this we have three strategies: (i) entire cluster/swarm of machines by omitting labels parameter in config creation, (ii) all nodes inside cluster/swarm using specific labels in configuration creation (only nodes/applications with specified labels will get message), and (3) using peer-to-peer approach. With the third strategy, users can deliver configuration in less network intensive manner.

Using `dissiminateConfig(labels, configList, percentage)` users can disseminate configuration in the cluster/swam only to nodes/applications that are desired using gossip protocol. For this, users provide a percentage of which nodes will get initial configuration details. tHis node might be desired one or not, nodes are picked by random, and message is only delivered to these nodes. On message arrival, nodes then disseminate configuration piggybacking on gossip protocols that already run inside cluster/swarm. When gossiping, nodes will disseminate configuration to the nodes that desired configuration is for. Notice that all configurations, before sent, are validated against already defined schema and only if validation passes configuration through, messages will be sent.

Using `getConfigTimeline(config)` users can get a timeline of both stand alone configuration, and configuration group changes over time. This view is particularly useful when one configuration (stand alone or as a group) is changed over time and something causes a problem. With this, users can easily detect what configuration caused a problem, when such a problem occurred and how to fix things.

2.5.16.3 Distribution

This tool is provided as open source, through the following repository:

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

The repository provides a simple README file that contains instructions on how to use it.

2.5.17 T-WP6-09 Telemetry Acquisition for Decentralised Systems for Containers

Another essential part of the runtime management of swarm systems, which is also essential to offline training of models using machine learning, or to evolve such models online, for instance through reinforcement learning, is to acquire information about the system at runtime. Such information is, at a first level, related with resource consumption, and the external perceptible state of different swarm elements. Similarly to the previous reported tool, the telemetry acquisition for decentralised systems tool reported here, while not being limited to, also takes advantage of containerization technology, which provides simple access to several runtime metrics out of the box, namely resource consumption.

2.5.17.1 Architecture and Design

This tool is materialised as a service designed for metrics acquisition, storage, and exporting to other platforms (e.g., Prometheus), and was designed with scalability and the cloud-edge continuum model in mind. Metrics are acquired from clusters/swarms of machines, collected, stored, and when needed exposed to other components to use the stored data for future benefits to other components in the TaRDIS toolbox. This service relies on open source tools (e.g., cAdvisor, Node Exported, Prometheus, Grafana, NAS, gRPC, etcd, etc.) for its development.

The system collects metrics from various places: (i) physical infrastructure nodes, (ii) applications running inside containers and (iii) other related metrics. To ensure that all metrics are collected, stored and used properly, collections of all groups are unified, enabling the correlation of different metrics at analysis time. To collect node metrics a *Node Exporter* is used. The *Node Exporter* exposes a wide variety of hardware- and kernel-related metrics. To collect metrics from containerized applications *cAdvisor* is used, *cAdvisor* is open source tool and industry standard for collecting containerized metrics. It is used in *Kubernetes* among many other places. *cAdvisor* collects metrics in a timespan of 1 minute locally. A third group of metrics include metrics which could be provided by the application itself, or even by internal components of the applications (such as decentralised membership or communication protocols). To this end, the integration with a different tool is essential. One such tool, for the Babel ecosystem, is reported further ahead in Section 2.5.18.

All metric groups are aggregated into a single element and sent to a storage service for further use (e.g., monitoring, dashboarding, alerting, processing, or machine learning activities). For metrics storage, processing, and exposing for other uses, *Prometheus* is used as a storage layer. *Prometheus* is an open source project and industry standard for collecting and storing metrics from distributed/decentralised applications. All collected metrics sent to from nodes are in *Open Metrics* format (which has become a standard way to store and share such metrics). This ensures that users can swap and use any tool in the future that is aligned with this format.

Collected metrics for infrastructure and applications can be visually shown using dashboards capable of reading Open Metrics data. For time being, all metrics are visually shown using open source tool *Grafana*, but can be easily swapped with any other tool capable of connecting to *Open Metrics* protocol aligned storage.

2.5.17.2 API

This tool provides the following API:

```
getMetrics(id) → timeseries

getNodeMetrics(id) → timeseries
```

Funded by the European Union

```
getApplicationMetrics(nodeId, namespaceId, appId) → timeseries

exposeMetrics(timestamp) → timeseries

exposeMetrics(start, end) → timeseries

customMetrics(data) → status

retentionPeriod(time, metric) → status

healthcheck(topic) → channel
```

Users are able to access stored metric data using a different set of available mechanisms with different granularities: (i) metrics per node using `getNodeMetrics(id)`, (ii) application metrics using `getApplicationMetrics(nodeId, appId)`, and (iii) cluster/swarm metrics using `getMetrics(id)`. For these to work, users must provide some of the identifiers like cluster/swarm id for the latter operation, node id for first operation or combination of node id, namespace id and application id in the particular case of the second operation. As a result, users will get metrics time series in *Open Metrics* format.

Results can be shown in a less detailed way using Command Line Interface (CLI) tool, or visual dashboard like Grafana. Stored metrics are exposed to others to utilise machine learning tools for example. For this purpose two options are available: (i) using `exposeMetrics(timestamp)` we can get data from the last synchronisation point. Users provide a timestamp of when was the last time that data was collected from the system, or (ii) using `exposeMetrics(start, end)` users can gata in between some time period which could be useful if tools using metrics data are missing some part of the entire time series. It is important to note that users can specify how long they want to store metrics data using `retentionPeriod(time, metric)` call.

For other metrics that need to be present in the general time series, users can use `customMetrics(data)` to store any valuable information that is not part of the node and/or application metrics. For example users can store metrics regarding communication protocols, membership protocols or any other valuable data that could serve the system to be better. The last api provided by the service is where `healthcheck(key)` users can subscribe to specific elements of the metrics acquisition subsystem to notify other interested parties when some of the data is changed (e.g., health check, alerting, auto scaling etc.). Users can subscribe to specific topics and as a result they get a channel to where events will be sent when the system gets a specific metric. This allows other sides to extend existing infrastructure for future extensions and capabilities.

2.5.18 T-WP6-10 Telemetry Acquisition for Decentralised Systems in Babel

While the previously presented tool allows to easily acquire and export telemetry information about the runtime environment of swarm elements (such as computational resource consumption, or the status of the application perceived externally), in many cases it is useful to combine such information with telemetry acquired from within the swarm application, be it the application logic itself, or even detailed telemetry of internal components that are part of the swarm application logic. For instance, to adapt the operation of a epidemic style application-level broadcast protocol, it could be useful to know, for a particular (or more likely several) swarm elements, how many redundant messages were received within a given time window, as this can be an indication that the *fanout* (i.e., the number of logical swarm neighbours to where each message received for the first time is forwarded) can be adjusted

(either increased or decreased) based on some mathematical model or some machine-learning derived model.

Unfortunately, and as far as we know, no development framework that can produce binaries ready for deployment (i.e., frameworks that are not simulators such as Peesim) provides support for the programmer to collect/manipulate such metrics at such a fine grained detail and easily export them (depending on the needs at runtime). To overcome this lack of support, we developed and integrated in Babel (the original Babel and the under development Babel-Swarm framework) mechanisms that empower developers to instrument their applications to monitor application/protocol specific runtime indicators, general hardware, and arbitrary metrics, while minimising the interference of these mechanisms with the code of distributed protocols and applications and ideally with minimal performance impact.

To achieve this, our proposal not only provides support for the most common metric types, such as, counters, gauges, and histograms, but also, offers sets of ready to use metrics depending on the type of protocol or application being developed. For example, if the developer is creating a variation of the Paxos agreement protocol, they may be interested in collecting metrics that are relevant for general agreement protocols, such as the average time or number of messages needed execute a round of agreement, and as such we want them to be able to have these metrics to be easily accessible for use without being a significant distraction in the development cycle

The collected metrics must then be exported, so they can be used to observe the behaviour of the system. These can be exported using a multitude of methods, such as exposing it through HTTP endpoints, exporting it to local log files, publishing them to a Message Queue to be processed, or exporting them to the tool presented in Section 2.5.17.

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

This solution will also include an application that serves as monitor to receive and display the exported metrics, which currently is centralised. We plan to soon produce solutions that can achieve this in a more decentralised fashion, that is therefore, more suitable for large scale decentralised swarms.

2.5.18.1 Architecture

In this section we briefly remind the reader of the architecture of this tool, which was more precisely detailed in Deliverable 6.1.

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

2.5.18.2 API

The API for the telemetry acquisition is here divided into the API for the metrics and the API for the Metrics Manager.

All metrics have a common API:

```
reset() → void

labelValues(labelValues) → void
```

The `reset` call resets the metric back to its initial value, this is called when exporters specify that metrics should be reset when they are exported.

The `labelValues` call is used when the Metric was initialised using label names; they are used to separate metric values, depending on the context. This call returns a simpler version of the metric which adheres to the API specified for the given metric type. This call must be used in metrics where label names were specified.
As an example, if we have a Counter that keeps track of the number of total requests made to another protocol, we may be interested in using labels to separate successful from unsuccessful requests.
In this specific example the Counter would be initialised with the label name “success” and when increasing our Counter we would need to specify a labelValue which, in this case, could be “true” or “false”.

Unless specified in their corresponding, Metrics are instantiated by supplying a name, the unit of measurement, and optionally the previously mentioned label names.

In order for users to instrument their protocols, they must use the API of the corresponding metric type.

The Counter has the following API:

```
inc(n)→ void

inc() → void
```

The first call would increase the counter by the specified number, whereas the second would increase the counter by one.

The Gauge has only a single call:

```
set(n)→ void
```

Which sets the Gauge to the specified value.

The Histogram adheres to the following API:

```
record(n)→ void
```

Which records the given value in the Histogram in the appropriate bucket, increases the number of values recorded and adds that value to the Histogram sum.

Since Histograms require buckets, this metric type needs buckets to be specified when it is instantiated, besides the other parameters common to all metrics.

The Metrics Manager provides the following API:

```
registerMetric(metric) → void

registerExporters(exporters) → void

startMetrics() → void
```

Users are able to create metrics of the available types (Counter, Gauge or Histogram), using the aforementioned APIs, which they can use to instrument the protocol they are developing. For this created metric to be collected and exported by the Metrics Manager the user needs to use the `registerMetric(metric)` call thus associating that metric with the ID of the protocol being developed.

For the metrics to be exported, so they can be used to get insights about the system once it is running, the user needs to register at least one Exporter, whose responsibility is to ask the Metrics Manager to sample the metrics so they can be exported. This is done by instantiating an Exporter, which can either be one of the ones provided out of the box, or one developed by the user for their specific use case, and after that using the `registerExporters(exporters)`call, which will register that exporter instance to the Metrics Manager.

Once the user has registered all the necessary exporters, they must call `startMetrics()` which will assign each registered Exporter to its respective thread, which in turn will start the metrics collection and exporting process.

2.5.18.3 Distribution

This tool is currently integrated within both the original Babel framework, and the under development Babel-Swarm framework. Hence developers making use of one of these frameworks have now access to these abstractions and functionalities out of the box.

Internally, we have been using this tool to evaluate our own work in developing the self-configuration, self-management, and security mechanisms in Babel-Swarm. We remind the reader that the features, architecture, implementation and preliminary evaluation of Babel-Swerm will be presented in detail in the upcoming Deliverable 6.2.

2.5.19 T-WP6-11 Babel Common APIs for Adaptive Protocols

To ensure that swarm systems can be enriched with self-management (approximating them of a complete autonomous system as originally envisioned in the seminal IBM manifest[53]) components within a swarm application must be possible to be managed by other components, being them internal to the process or potentially external i.e., some external controller that runs on his own process - potentially on a different machine - that perform analysis of telemetry information, and based on high level goals, a fine-grained policy, or some machine learning static or dynamic model - issues adaptation commands to swarm elements, or a particular protocol within a swarm element. In general we refer to such a component that performs this type of task as an autonomic *manager*.

Focusing on the particular case of managing the execution of different distributed protocols, in particular by modifying runtime parameters that govern their operation (e.g., the number of neighbours kept by each swarm element at the level of a decentralised membership protocol, or the fanout value used by some gossip-based dissemination protocol), a possibility to support this would be to allow the developer of each individual protocol to specify its own adaptive API, meaning the Babel events that should be used by a component that is issuing commands to change runtime parameters to interact with that protocol. This however bring several disadvantages, the first is that the process of defining (manually or even assisted by some intelligent IDE as the one being developed in TaRDIS) several Babel events to control parameters is time consuming. More importantly, if each protocol features a set of management events that are different from other protocols it breaks the modular approach of a framework such as Babel, and would lead to programmers having to implement their specialised controller for each different combination of protocols such that the controller is aware of the API exposed by those protocols.

To avoid this scenario, we have defined a library that defines an extensible API (i.e., Babel events) that can be imported by developers of protocols to expose their management operations in an unified way. This allows to develop manager components that are more[54] protocol agnostic, and reusing these components to build reliable and self-adaptive swarm systems.

Actually, this library not only exposes the events that allows an autonomic manager to issue adaptation commands, but also specify an annotation that allows a developer to tag some class property of a protocol as being a runtime parameter that can be modified at runtime, and also events that an autonomic manager can use to request to a protocol that support autonomic management (or in other words an Adaptive Protocol) the runtime parameter that it has that can be modified at runtime.

Below we detail the API of this tool and provide the mechanisms to import it into a project. Notice that this tool is still evolving since we have only recently started to address this goal of tardis within the actions of WP6. Moreover, we expect that only (adaptive) protocols and autonomic managers developers will have to use this API, although it is possible that, in some cases, the application logic might want to modify the runtime configuration of some protocol, which this API also simplifies.

---

[53] Horn, P. (2001) Autonomic Computing: IBM's Perspective on the State of the Information Technology.

[54] Here we say more because naturally, in cases where the adaptation strategy is highly entwined with the protocol operation, it can happen that a manager component will be designed specifically to that protocol. This said there are clearly examples where the manager logic can be protocol agnostic, for instance, the simple manager that we developed to manipular at runtime the number of neighbours of a random overlay network and the fanout of a gossip-based dissemination protocol is valid for most combinations of these class of protocols.

2.5.19.1 API

Since this is a tool designed to enrich the operation, and simplify the use of Babel, its API is based on a set of Babel events that we now briefly describe.

We should note that the original Babel framework does not support adaptive protocols, hence this library is only compatible with the new Babel-Swarm framework that is currently under development.

Requests

- IncreaseNumberNeighbors, which indicated to an adaptive membership protocol that it should increase its current number of (direct) neighbours. By default this increment is of one unit, but the developer might provide a specific (positive and greater than zero) number.
- DecreaseNumberNeighbors, which indicated to an adaptive membership protocol that it should decrease its current number of (direct) neighbours. By default this decrement is of one unit, but the developer might provide a specific (positive and greater than zero) number.
- GetAdaptiveFieldsRequest, which requests an adaptive protocol the list of all runtime parameters that it features that can be changed.
- Reconfigure, which includes a map of parameters to be modified by the protocol. In this map the key is a String that identifies the name of the parameter to be modified, while the value (of type Object) represents the value that should be attributed to that parameter. The reader should note that handling this operation might not be simply invoking the setter for the variable storing this runtime parameter, since the protocol might need to change other aspects of its operation for a single parameter change.

Replies

- GetAdaptiveFieldsReply should be generated by an adaptive protocol in response to the GetAdaptiveFieldsRequest. It contains a map that represents both all the runtime parameters that can be changed and the current value of those values (using types similar to those used by the map in the corresponding request.

Notifications

- ReconfigurationSucceeded, is a notification that can optionally be generated by a protocol after it executes a Reconfigure request successfully. An individual notification is generated for each runtime parameter that is modified successfully (which means that a single Reconfigure request can generate several of these notifications). The notification carries information about the name of the parameter, the new value, and the name of the protocol that issued the notification.
- ReconfigurationFailed, is a notification that should be generated by a protocol if it cannot modify the value of a runtime parameter within a Reconfigure request. Similarly to the notification above, one notification should be generated for each individual runtime parameter that was not possible to be modified. The elements within this notification are similar to those of the ReconfigurationSucceeded notification.

2.5.19.2 Distribution

The code of this tool is open source and available here:

https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/babel-adaptivecommons

Similarly to other tools that integrate with the Babel ecosystem it can be imported by developers using maven, through the following directives in the pom.xml file:

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>

<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-adaptive-commons</artifactId>
    <version>[1.1.1,)</version>
</dependency>
```

2.5.20 T-WP6-12 Decentralised Membership Protocols for Swarms

As stated before in this document (and previously in Deliverable 6.1) decentralised membership protocols - that usually are materialised by an overlay network - are at the foundation of the operation of most swarms, at least those that are large enough that justifies avoiding the overhead and scalability penalty that derives from tracking and keeping up-to-date a large number of members that change frequently.

In addition to the Epidemic and Scalable Global Membership protocol that we designed (on top of some of the abstractions reported further ahead in the document), and that offers an eventually correct global membership to every element of the swarm, we have implemented other membership abstractions. In the following we briefly describe these membership abstractions.

HyParView

We have implemented HyParView[55] which is still, as far as we know, the random overlay network with the highest fault-tolerance, being able to recover from large correlated simultaneous failures as high as 80% of all nodes in the network. This implementation follows the specification in the original paper. The protocol combines two independent partial-views that are managed by different strategies for different purposes.

In more detail, it combines a small partial view that is managed in a reactive fashion, meaning that the contents of that view (i.e., the neighbours) only change in reaction to external events, such as a node failing or a node joining the network. Elements of this view are monitored using permanent TCP connections. A second and large partial view, named the passive view, is managed using a cyclic strategy, where the contents of those views are continually updated to reflect changes in the system membership through periodic random walks issued by each node. Only neighbours in the active view are used to exchange information (and only those trigger NeighborUp and NeighborDown notifications), while nodes in the passive view are used to expedite the recovery of active views when neighbours

[55] 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.

are suspected as being failed. It was implemented as a Babel (the original framework) protocol.

In addition to this implementation that follows the original specification of the protocol, we have also implemented three additional variants of the protocol that leverage on the different functionalities provided by the new version of Babel that we named Babel-Swarm. The reasoning for these variants is two fold. First it allowed us to validate and experiment with each of the functionalities of Babel-Swarm in isolation. The Second reason is that it provides additional flexibility to programmers using the TaRDIS Babel ecosystem to rely on a variant of the protocol featuring specific properties. We discuss these below.

HyParView-With-Discovery

This variant of HyParView relies on the mechanisms in Babel-Swarm to automatically discover a contact node in the local network using either the Multicast-based or Broadcast-based discovery mechanism. Other than that the implementation adheres to the original protocol specification.

HyParView-Autonomic

This variant of the protocol, in addition to supporting the automatic discovery of contact nodes, also features the mechanisms related with automatic configuration of protocol parameters (either taking advantage of other nodes in the network or DNS records) and also allows parameters that govern the operation of the protocol (e.g., the size of partial views) to be adapted at runtime.

Secure-HyParView

This variant of the protocol, in addition to supporting the automatic discovery of contact nodes, features security mechanisms provided by Babel-Swarm. In particular the identifiers of nodes are enriched with a unique identifier associated with a self-signed certificate that is presented to prospective neighbours. This allows them to authenticate nodes when they establish a TCP connection. Furthermore, this implementation relies on secure (point-to-point) Babel communication channels, either based on TLS or providing authentication or integrity of messages.

X-BOT

We also implemented X-BOT[56] (for the original Babel framework). X-BOT is a protocol that manages a random overlay network that biases the resulting topology of the overlay to promote links with a given property. In particular this implementation focuses on promoting links with lower latency. This implementation follows the original paper algorithm that has two main interesting aspects: i) bias is conducted through a 4-node coordinated strategy where iteratively, a node starts an optimization round with 3 other nodes to switch two existing overlay links by two other with lower latency; and ii) to protect global connectivity, each node keeps the link with highest latency unbiased, which was shown to be enough to ensure connectivity.

[56] 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.

2.5.20.1 API

All protocols discussed above leverage on the Babel-Protocol-Commons API for membership API previously presented in Section 2.5.9.

Additionally, the HyParView-Autonomic variant takes advantage of the API for self-adaptive protocols described in Section 2.5.19.

We omit these APIs since they were already presented.

2.5.20.2 Distribution

All of the previously presented membership protocols are provided both as open source code and as an independent java library (jar) that can be easily imported using maven. All of these tools are available to one of our maven repositories, that as presented multiple times throughout the document, can be configured in the pom.xml file (used by maven) using the following directives:

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>
```

In the following we present the repository and the maven dependency that a developer can use to respectively, access the code, and import the dependency in maven.

Notice that while the original version of HyParView and the X-BOT protocols only operate in the original Babel framework, the remaining versions of HyParView only operate in Babel-Swarm. This happens because these versions are explicitly making use of the new abstractions and support provided by Babel-Swarm, namely automatic configuration (with contact discovery), autonomic management, and security abstractions respectively.

As explained above, the different versions HyParView empower the developer to explicitly pick (at the dependency level) the functionalities that his application requires. As one can expect the main logic of the protocol is the same, and hence the code-bases are more than 50%-60% similar.

HyParView (for Babel only)

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

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-autonomic-hyparview</artifactId>
    <version>[1.1.2,)</version>
</dependency>
```

HyParView-With-Discovery (for Babel-Swarm only)

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>hyparview-discovery</artifactId>
    <version>[0.1.18,)</version>
</dependency>
```

HyParView-Autonomic (for Babel-Swarm only)

https://codelab.fct.unl.pt/di/research/tardis/wp6/hyparview-autonomic

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-autonomic-hyparview</artifactId>
    <version>[0.1.10,)</version>
</dependency>
```

Secure-HyParView (for Babel-Swarm only)

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

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocols-hyparview-secure</artifactId>
    <version>[0.1.20,)</version>
</dependency>
```

X-BOT (for Babel only)

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

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-autonomic-hyparview</artifactId>
    <version>[0.0.1,)</version>
</dependency>
```

2.5.21 T-WP6-13 Decentralised Communication Protocols for Swarms

We have implemented a total of four communication abstraction protocols that provide some point-to-multipoint abstraction. We now briefly discuss these protocols.

Flood Broadcast

The flood broadcast protocol is an application-level broadcast protocol that operates on top of a random overlay network, where all links of the overlay are flooded (i.e., used to retransmit a message) when a message is broadcasted. Duplicates are detected by the protocol and are not processed nor delivered to the application layer.

Eager Gossip Broadcast

The eager gossip protocol is another epidemic application-level broadcast protocol that differs from the flood by having each node only forwarding a message that is disseminated and received for the first time to a subset of the unstructured overlay neighbours (a parameter usually called *fanout*). This protocol relies on the infect and forget pattern that was proposed several years ago.

One-Hop Broadcast

This is a simple protocol that also operates on top of an unstructured overlay network, and that simply disseminates messages to all direct neighbours of the sender. Contrary to the previously presented protocols, in this case the message is not propagated again, meaning that a message reaches only the direct neighbours of the sender. This is useful, for instance, to propagate information control to direct neighbours to govern localised decisions.

Anti-Entropy

This is a novel protocol that we have developed and operates in cooperation with other dissemination protocols such as the eager push gossip described above. The protocol operates by keeping at each node a buffer with (recently) delivered messages. Periodically, each node picks a random neighbour and sends to it a summary (using a bloom filter) summarising messages in this buffer. The receiver of this message looks at his own local buffer, and if there is a message there not present in the received bloom filter triggers a retransmission at the gossip protocol level.

Evidently, to avoid messages to circulate in the network forever and to limit the amount of memory consumed by the buffer, the protocol features mechanisms for garbage collection and a mechanism where some messages in the buffer are not retransmitted. This protocol will be further detailed on the upcoming Deliverable 6.2.

2.5.21.1 API

All protocols discussed above leverage on the Babel-Protocol-Commons API for dissemination and communication API previously presented in Section 2.5.9.

2.5.21.2 Distribution

All of the previously presented communication protocols are provided both as open source code and as an independent java library (jar) that can be easily imported using maven. All of these tools are available to one of our maven repositories, that as presented multiple times throughout the document, can be configured in the pom.xml file (used by maven) using the following directives:

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>
```

In the following we present the repository and the maven dependency that a developer can use to respectively, access the code, and import the dependency in maven. Notice that with the exception of the Flood Broadcast protocol, whose code is only compatible (for now) with the original Babel framework, the remaining protocols have a different dependency for the original Babel framework and another for the under-developments Babel-Swarm. The reason for this is twofold, on the one hand Babel-Swarm is using java version 21, whereas the original Babel uses Java version 8, which on its own would not require this duplication,

although under Java 21 we can use (in the future) constructs of the language that are more modern and performant. The main reason for this duplication is that these protocols depend on the Babel-core, which is different between the two versions of the framework. To ensure that dependencies are correctly handled, this requires that all protocols designed to operate in Babel-Swarm, are materialised on a different artefact that explicitly depends on that Babel core version instead of the old one.

FloodBroadcast (for Babel only)

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

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-floodbcast</artifactId>
    <version>[1.0.0,)</version>
</dependency>
```

Eager Gossip Broadcast (for Babel)

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

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-eagerpushgossip</artifactId>
    <version>[1.1.5,)</version>
</dependency>
```

Eager Gossip Broadcast (for Babel-Swarm)

https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/protocols/eagergossipbroadcast

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-eagerpushgossip-j21</artifactId>
    <version>[1.1.5,)</version>
</dependency>
```

One-Hop Broadcast (for Babel-Swarm)

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

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-onehopbroadcast-j21</artifactId>
    <version>[0.0.5,)</version>
</dependency>
```

Anti-Entropy (for Babel)

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

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-antientropy</artifactId>
    <version>[0.1.4,)</version>
</dependency>
```

Anti-Entropy (for Babel-Swarm)

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

```
<dependency>
    <groupId>pt.unl.fct.di.novasys.babel</groupId>
    <artifactId>babel-protocol-antientropy-j21</artifactId>
    <version>[0.1.4,)</version>
</dependency>
```

2.5.22 T-WP6-14 Decentralised Estimator of Swarm size

As part of the TaRDIS efforts to achieve self-management on swarms, we recognize the need to collect telemetry information from individual swarm elements (both externally - see Section 2.5.1 - but also from internal components of a swarm application - see Section 2.5.18) but also information about the execution environment. To start exploring this venue we consider the swarm size (i.e., the number of elements, or processes, currently active in a swarm). This is challenging because in a decentralised system no node has a complete view of the system state.

While we could use the Epidemic and Scalable Global Membership protocol (see Section 2.5.10) to obtain this information, we believe that in highly dynamic and large-scale systems this has a non-negligible overhead. To overcome this limitation we started to study less computational, memory, and bandwidth intensive alternatives that can enable each node in the swarm to estimate the size of the swarm. We found several alternatives although not many can cope with dynamic systems.

An interesting alternative was found in a protocol based on the Random Tour method[57] that we have implemented. This protocol simply propagates a message with an increasing counter at each hop, based on the current's node degree. The message keeps traversing the network until it finds its way back to its originator, who then infers some aggregation function based on that counter. This means that if we try to calculate a SUM and each node contributes with a value of 1, we can estimate the network size.

To avoid fluctuations in the system due to imprecise estimates, we maintain a window with the last 10 estimates, and report an average of this to the autonomic controller. This has the additional benefit that it avoids adaptations to happen too early after a node joins the system (since no adaptation happens until 10 estimates are completed).

[57] Massoulié, L., Merrer, E., Kermarrec, A.M., Ganesh, A.: Peer counting and sampling in overlay networks. pp. 123–132 (07 2006). https://doi.org/10.1145/1146381.1146402

2.5.22.1 API

This protocol operates on top of an unstructured overlay, and it emits a single synchronous notification that carries the average of the last 10 estimates made by the local protocol based on a sliding window of measurements as discussed above.

The notification is called OverlaySize which only carries the current estimate.

2.5.22.2 Distribution

The code for this protocol is provided as open source in the following repository:

https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/utils/network-size-estimation-rt

Similarly to other Babel protocols, the protocol can be imported using maven through thee following directives:

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>

<dependency>
    <groupId>pt.unl.fct.di.novasys.babel.utils</groupId>
    <artifactId>random-tour</artifactId>
    <version>[0.0.1,)</version>
</dependency>
```

2.5.23 T-WP6-15 Localised Simple Static Autonomic Swarm Manager

Still focused on showing the viability of achieving autonomic control of protocols supporting a swarm application, in particular using a static (i.e., analytical) model and only localised information, and featuring a local adaptation strategy, we implemented a simple autonomic manager.

The reader should note that we made efforts to ensure a decoupling between telemetry acquisition protocols and the autonomic monitor to allow for different strategies for the collection of telemetry to be employed. For this to be viable, metric collection protocols follow a predefined interface consisting of messages and requests, which can be extended if necessary. The autonomic controller waits for (asynchronous) metric announcements that support decision-making, which are sent by those protocols. In this particular case the estimates of the swarm size provided by the protocol described above.

Our simple controller was designed such that after receiving an estimation of the network size, it decides, based on a static and simple model derived from the theory of epidemics, which reconfiguration can be made to the number of neighbours of an unstructured overlay network. The reconfiguration to be done incrementally to minimise unexpected consequences and moreover, our controller relies on a minimum amount of time between issuing reconfiguration requests to allow the system to stabilise from the previous change. This means that, in particular, even if the estimate of the network size could justify an increase in the number of neighbours by two, we do this increment one neighbour at a time (and similarly to the fanout of the epidemic broadcast protocol).

The HyParView-Autonomic protocol, described above, is, so far, the only protocol in our toolbox compatible with this methodology.

2.5.23.1 API

The autonomic controller takes advantage of the API for self-adaptive protocols described in Section 2.5.19. We omit that API here for conciseness.

2.5.23.2 Distribution

This autonomic manager is provided as open source in:

https://codelab.fct.unl.pt/di/research/tardis/wp6/babel-swarm/simple-autonomic-controller

It can be easily imported by a swarm application developed in Babel using maven through the following directives in the pom.xml file.

```
<repositories>
    <repository>
        <id>novasys-mvn</id>
        <url>https://novasys.di.fct.unl.pt/packages/mvn</url>
    </repository>
</repositories>

<dependency>
    <groupId>pt.unl.fct.di.novasys.babel.autonomic</groupId>
    <artifactId>simple-autonomic-controller</artifactId>
    <version>[0.0.1,)</version>
</dependency>
```

3 Requirements Update

3.1 Summary of the Previously Defined Requirements

In deliverable D2.2, use case providers described in detail each use case and specific scenarios for each of them. This was a first step towards defining a set of functional and non-functional requirements for the TaRDIS toolbox capturing the needs of each use case. In total, 58 requirements were provided and a summary of these is provided in the table below.

Summary of requirements defined by each use case provider in Deliverable 2.2

Type of Requirement	**Use Case Provider**			
	EDP	TID	GMV	ACT
Functional	3	8	10	23
Non-Functional	2	4	2	6

The set of aforementioned requirements generated a set of toolbox requirements that was compiled within work packages 3–6 based on the use case requirements and by means of a collective effort within the consortium. This set of toolbox requirements is crucial for ensuring a bi-directional mapping between the tools and the needs of the use cases. Having that in mind, all requirements included a forward and backward traceability, i.e., use case requirements were mapped to toolbox requirements and the other way. Moreover, toolbox requirements were flagged with different levels of priority, indicated dependency with other requirements, and lastly, were mapped to potential KPIs.

Given that this deliverable provides a clearer view of the different tools built so far, their functionality and their interplay with other tools, we believe that an update of some requirements might be necessary. Such updates might include small changes (e.g., concerning the level of priority) or bigger changes such as the elimination of previously defined requirements of the creation of new ones. In what follows, we provide an update on the requirements for each use case with a thorough justification. Of course, these updates affect the toolbox requirements, and we will discuss this at the end of this Section.

3.2 Updates on Use Case Requirements

In this section we report on changes to the direct requirements from use case implementations that became necessary as we progressed with the implementation and learnt more about the TaRDIS toolbox while it was taking shape.

3.2.1 Energy Multi-Level Grid Balancing

The Energy Multi-level Grid Balancing, is an energy use case focused on the participation of consumers and producers inside an energy community. This concept follows a multi-layer edge-fog-cloud approach parallel to the LV-MV-HV to ease the creation of an energy community, enable intra/inter-community communication, address low-voltage grid capillarity, and provide grid monitoring and node orchestration. Ultimately, aims to improve efficiency, balance grid nodes, and increase the share of RES.

Previously on D2.2 it was presented RF-EDP-01:

Original version of the functional requirement RF-EDP-01 (source D2.2)

ID	RF-EDP-01	Priority	Must
Name	Exchange agreement between prosumers		
Description/Rationale	Description: The system must broadcast accurate energy consumption forecasts and securely finalise real-time peer-to-peer (P2P) agreements among producers and consumers. These features are essential for efficient and secure intra-community energy exchange. Rationale: Accurate Forecasting enables users to optimise energy use within the community, ensuring a reliable energy supply. Real-Time P2P Agreements enhances responsiveness, allowing quick adaptation to changing energy demands within the community. Energy Efficiency minimises grid imbalances and reduces energy losses, contributing to a sustainable and cost-effective intra-community energy exchange.		
Dependency	No dependency with other requirements		
Traceability (backward)	EDP use case Scenario 5 and Scenario 6 UC-01-Scenario 5 – Running in Normal operation mode. UC-01-Scenario 6 – Running with faults.		
Traceability (forward)	WP4-Requirements WP5-Requirements WP6-Requirements		
Linked KPIs	K-B-01: programmer effort for overlay network K-B-02: network bandwidth used K-B-13: latency at interested peers K-B-17: security verification effort K-U-02		

This requirement was now updated in two parts 01a and 01b as shown below.

Derived functional requirement RF-EDP-01a

ID	RF-EDP-01a	Priority	Must
Name	Community acceptance and network registration		
Description/Rationale	Description: The Community Operator (CO) must enable new community members through the swarm membership tool (T-WP6-02) and rely on software-defined networking (T-WP4-04) to use the data network with the other community peers. The network runs over a GPRS network settled over the mobile operator APN. Rationale: Secure settings of the system		

Dependency	No dependency with other requirements
Traceability (backward)	EDP use case Scenario 5 and Scenario 6 UC-01-Scenario 5 – Running in Normal operation mode. UC-01-Scenario 6 – Running with faults.
Traceability (forward)	WP4-Requirements WP5-Requirements WP6-Requirements
Linked KPIs	K-B-01: programmer effort for overlay network K-B-02: network bandwidth used K-B-13: latency at interested peers K-B-17: security verification effort K-U-02

Derived functional requirement RF-EDP-01b

ID	RF-EDP-01b	Priority	Must
Name	Exchange agreement between prosumers		
Description/Rationale	Description: The system must enable consumers to broadcast information (T-WP3-03), namely accurate energy consumption forecasts, using Protocols (T-WP5-03/06) and peer-to-peer (T-WP6-04) agreements. The messages comprise packets of around 10kB with all the agreement settings. The forecasting of the energy consumption can be performed with a suitable ML model for time-series forecasting algorithm (as described in requirement D2.2-RF-WP5-FL-ALG-05). The training of the ML models (each peer/prosumer has its own individual model) can be performed in a federated manner inside the Energy Community by using a Federated learning framework, as described in D2.2-RF-WP5-FLALG-01. The online ML model inference can leverage lightweight ML techniques (D2.2-RF-WP5-FL-ALG-06) to reduce the energy consumption related to the model inference, while their forecasting performance can be also evaluated (D2.2-RF-WP5-FLALG-04). Rationale: Real-Time P2P Agreements enhance responsiveness, allowing quick adaptation to changing energy demands within the community.		
Dependency	No dependency with other requirements		

Traceability (backward)	EDP use case Scenario 5 and Scenario 6 UC-01-Scenario 5 – Running in Normal operation mode. UC-01-Scenario 6 – Running with faults.
Traceability (forward)	WP4-Requirements WP5-Requirements WP6-Requirements
Linked KPIs	K-B-01: programmer effort for overlay network K-B-02: network bandwidth used K-B-13: latency at interested peers K-B-17: security verification effort K-U-02

The desegregation allows for the refinement and detailing of this requirement in alignment with the TaRDIS tool. Additional updates to the requirements will be made as the tools mature, and the tools should be reusable for different applications, further updates on the application to the energy UC will be in D7.2.

3.2.2 Privacy-preserving learning through decentralised training in smart homes

Telefonica's use case revolves around smart homes where different devices are part of an automated system that monitors and/or controls home attributes such as lighting, heating, entertainment systems, and smart appliances. Through federated learning (FL), these devices train deep neural networks based on their local datasets. This use case focus is also on addressing two major challenges in this setting: device heterogeneity, where some devices might be unable to train a model locally, and privacy, since some of the data used for FL expose individual user behaviour/preferences. These challenges will be addressed by enabling Split Learning (SL), and privacy in (Hierarchical) FL. Details on these settings were provided in Deliverable 2.2, along with a set of use case requirements. We take the opportunity now to clarify aspects that were not fully defined at the time, or that, in hindsight, might not have been clearly stated in the previous deliverable: we plan to consider a decentralised (partially or fully) version of the two paradigms where nodes that are higher in the hierarchy may communicate with each other for aggregation or privacy noise injections, resulting in lower (or zero) dependency to central nodes/infrastructures such as the cloud.

The requirements RF-TID-02 and RF-TID-04 were defined in Del. 2.2 in order to ensure security and privacy in cross-app training under the Federated Learning (FL) paradigm. In cross-app training, data generated through different apps on the same device can be merged to train a single neural network model in a distributed way in a collaborative way with other devices. As an example, one can consider the viewing history or movie ratings recorded on the Netflix mobile app and on the Disney+ app on the same phone. This data could be merged to train an "universal" model for a recommender system that could be used by both applications/services.

The updated versions of the requirements RF-TID-02 and RF-TID-04 differ from the original versions with respect to the priority level. In particular, as shown in the tables below, the priority has been now set to "Could", i.e., a property that would be nice to have- if possible, as opposed to "Should" that was indicated previously.

The priority level was updated because, unlike security in peer-to-peer communications and privacy in FL, in the cross-app setting, the security and privacy is at input data level and

affects how this input data affects the trained model. Furthermore, this setting assumes that two (or more) applications (and hence, their owners) are willing to collaborate and share their data. In practice, however, this might not be the case: one can think of the example of the Netflix and Disney+ applications, where these competing streaming platforms might not be willing to share data between them, even if that would allow both to provide better recommendations to their users, and hence increase retention (since a user that spends more time ni Netflix has less incentive to try or use Disney+ and vice versa). In fact, this opens a new direction of research that TID will try to pursue eventually, regarding the incentives of such apps to share data. We believe that these requirements would be, thus, of higher importance after a theoretical analysis of such scenarios.

Updated version of the functional requirement RF-TID-02

ID	RF-TID-02	Priority	**Could**
Name	Secure communications between applications (for cross-app training)		
Description/ Rationale	Protection against security threats such as data poisoning that can have an impact on the model (e.g., introduce bias)		
Dependency	No Dependency with other requirements		
Traceability (backward)	UC-02-Scenario 1: Presence of hierarchy in the system UC-02-Scenario 2: Split learning in the presence of devices with limited computation/memory capacity		
Traceability (forward)	WP6-Requirements		
KPIs	K-B-17: security verification effort		

Updated version of the functional requirement RF-TID-04

ID	RF-TID-04	Priority	**Could**
Name	Privacy of application data (for cross-app training)		
Description/ Rationale	Preservation of privacy of each application's data (given a possible app conflict of interest among different applications)		
Dependency	No Dependency with other requirements		
Traceability (backward)	UC-02-Scenario 1: Presence of hierarchy in the system UC-02-Scenario 2: Split learning in the presence of devices with limited computation/memory capacity		
Traceability (forward)	WP6-Requirements		
KPIs	K-B-09**:** FL privacy		

Next, we update the functional requirement RF-TID-03 that was previously named “Privacy of FL clients (for cross-device training)”. After revising the set of the requirements defined in Del. 2.3, we found that this requirement was not described in a very clear way and, for this reason, we renamed it and updated its description, as shown in the table below.

Updated version of the functional requirement RF-TID-03

ID	RF-TID-03	Priority	Must
Name	Privacy or anonymity in the training process (for cross-device training)		

Description/ Rationale	Ensure privacy preservation of the global ML model or/and anonymity or privacy of the participants in the training process (e.g., devices in federated or split learning).
Dependency	No Dependency with other requirements
Traceability (backward)	UC-02-Scenario 1: Presence of hierarchy in the system UC-02-Scenario 2: Split learning in presence of devices with limited computation/memory capacity
Traceability (forward)	WP3-Requirements WP6-Requirements
KPIs	K-B-09**:** FL privacy

Next, we update the functional requirement RF-TID-06 that was previously named “Workflow orchestration”. In particular, as shown in the table below, it is now renamed and its description was updated in order to better capture the use case need for an optimization module/component in the training system. This module might be based on available profiling data (e.g., profiled training times or memory footprint) in order to solve, in an offline or online manner, an optimization problem related to the workflow of the training operations and the management of the available resources (memory, communication channels, etc.). This is particularly useful in scenarios where the system is mostly stable and changes do not occur often.

Updated version of the functional requirement RF-TID-06

ID	RF-TID-06	Priority	Should
Name	Optimization of training operations and of the available resources		
Description/ Rationale	Need for a module/component that, based on the system’s characteristics and available (profiling) data, solves an optimization problem that concerns the workflow of communications among the participants and/or memory management of helpers and/or of the splitting policy in the case of Split Learning. This optimization problem would have been defined in advance (i..e, objective function, variables, model used) and will have as input the system’s characteristics.		
Dependency	RF-TID-05, RF-TID-07, RF-TID-08, RNF-TID-04		
Traceability (backward)	UC-02-Scenario 2: Split learning in the presence of devices with limited computation/memory capacity		
Traceability (forward)	WP3-Requirements WP4-Requirements WP6-Requirements		
KPIs	K-B-06: FL CPU usage for training, K-B-07: FL training latency, K-B-08: FL storage/RAM requirements per node, K-B-11: scalability, K-B-02: network bandwidth used K-U-04		

Finally, we update the requirement RF-TID-08 previously named “Helpers in Split Learning”. After revising the set of the requirements defined in Del. 2.3, we found that this requirement was not described in a very clear way and, for this reason, we renamed it and updated its description, as shown in the table below.

Updated version of the functional requirement RF-TID-08

ID	RF-TID-08	Priority	Must
Name	Allow hierarchies in the system		
Description/ Rationale	Ensure the existence and maintenance of helpers in split learning and of trusted entities in hierarchical FL. This might imply deciding which participant (clients, servers, etc.) qualifies to act as a helper that can help with the computations or as a trusted entity to inject differential privacy noise. This decision/election may depend on the resources available at each participant or other system’s characteristics.		
Dependency	Depending on the implementation this could be related to RF-TID-05 and RF-TID-06		
Traceability (backward)	UC-02-Scenario 1: Presence of hierarchy in the system UC-02-Scenario 2: Split learning in presence of devices with limited computation/memory capacity		
Traceability (forward)	WP5-Requirements WP6-Requirements		
KPIs	K-B-06: FL CPU usage for training, K-B-07: FL training latency, K-B-08: FL storage/RAM requirements per node		

To conclude, we made minor revisions on 5 requirements for this use case. These revisions focused on either the priority level of the requirements or their descriptions. A total of 12 requirements are defined for this use case, including 8 functional and 4 non-functional requirements.

3.2.3 Distributed navigation concepts for LEO satellite constellations

Based on deliverable D3.1 (First report on TaRDIS models and APIs), it was stated that the GMV use case will produce two applications:

1. A reference sequential simulation of the decentralised orbit determination (D3.1 Section 4.3.1), written mainly using Matlab and Python. This application will use the TaRDIS AI/ML APIs in combination with well-established modelling and simulation techniques adopted in the space industry.

2. A distributed simulation (D3.1 Section 4.3.2) that will additionally leverage the TaRDIS APIs for communication and distribution, to simulate a swarm of satellites performing decentralised ODTS by modelling each satellite as an independent distributed application instance.

An initial set of requirements taking into account these objectives were presented in report D2.2, also considering the different scenarios of interest that were defined for the GMV use case. Although the main scenario corresponds to the achievement of a distributed Orbit Determination and Time Synchronisation (ODTS), in which each satellite obtains its own

navigation solution in the most autonomous and robust way possible, other secondary scenarios were defined that are of great interest for the continuation of the previous one.

The first of these is based on the importance of the orchestration of the connections between the different satellites of the swarm by means of inter-satellite links (ISLs). The scheduling of these connections has key effect on the ODTS process, since the heterogeneity of the information fed to each of the nodes depends on it, as well as the geometry of the connections: the more diverse (different peers) and in more directions (network topology) the measurements made by each satellite through ISL, the better performance can be achieved in the ODTS process.

The last scenario details the importance of selecting a good tuning of the parameters that characterise the filter used for the ODTS resolution. The performance of the algorithm is highly dependent on the large set of input parameters that feed it, such as the different uncertainties that characterise the dynamic model, measurements and states. Therefore, the need for a multi-objective optimization of this problem was introduced, which allows to customise the filter as closely as possible to reality.

The progress in the development of the use case since report D2.2 has caused some updates regarding these last two scenarios, and the requirements that had been defined in relation to them (RF-GMV-06 and RF-GMV-08). While the initial priority was to optimise the filter tuning, we have finally opted for a change of direction towards the dedication of effort to assess the ISL scheduling problem. The influence of this on the final objective of realising an autonomous, resilient and robust ODTS is very strong, as has been proven by tests carried out with the tools developed for the baseline of the use case. The TaRDIS tools could help considerably to improve the development that has been done on this problem for the baseline, which is of greater importance than initially foreseen, and therefore GMV believes it is appropriate to modify the priority levels of the two aforementioned requirements. As shown in the tables below, RF-GMV-06 related to the tuning scenario has been now set to “Could” as opposed to “Must”, while RF-GMV-08 related to the scheduling problem has been changed to “Must”, instead of “Could”.

Updated version of the functional requirement RF-GMV-06

ID	RF-GMV-06	Priority	Could
Name	Capability to perform multiple simulations in parallel		
Description/ Rationale	It must be possible to simulate multiple scenarios in parallel (multiple simulations with different input settings, for instance initial errors, measurements noise, measurements frequency, process noise, propagator settings, etc). All these settings shall be configurable in the ODTS simulator.		
Dependency	No dependency with other requirements		
Traceability (backward)	UC-03-Scenario 3 -		
Traceability (forward)	WP6-Requirements		
KPIs	N/A (Validated in WP7 demonstrations)		

Updated version of the functional requirement RF-GMV-08

ID	RF-GMV-08	Priority	Must
Name	Optimization of the Inter-satellite Link connectivity scheme		

Description/ Rationale	The inter-satellite link scheduling algorithm shall provide an optimal connectivity scheme based on satellite visibilities at a certain time, therefore maximising navigation accuracy ensuring reliability and resiliency.
Dependency	No dependency with other requirements
Traceability (backward)	UC-03-Scenario 2-
Traceability (forward)	WP5-Requirements
KPIs	N/A

Additionally, after reviewing the requirements defined in report D2.2, we found that some of them needed updating or slight modifications in their descriptions and titles to be clearer. Such is the case of RF-GMV-07, RF-GMV-09 and RF-GMV-10 shown below, which have been reformulated to emphasise respectively that the aim is to reduce the number of operations and computational cost compared to the algorithms used in the baseline and to clarify the robustness property of the ODTS algorithm/ML model.

Updated version of the functional requirement RF-GMV-07

ID	RF-GMV-07	Priority	Should
Name	Orbit propagation algorithm efficiency		
Description/ Rationale	The orbit propagation module execution should be optimised (i.e. the number of operations should be reduced, as well as the computational resources).		
Dependency	RF-GMV-09		
Traceability (backward)	UC-03-Scenario 1 -		
Traceability (forward)	WP3-Requirements WP5-Requirements		
KPIs	K-U-06: Reduction of the use of computational resources.		

Updated version of the functional requirement RF-GMV-09

ID	RF-GMV-09	Priority	Must
Name	ODTS algorithm/ML model efficiency		
Description/ Rationale	The ODTS process executed by each node shall be optimised (i.e. the number of operations should be reduced, as well as the computational resources).		
Dependency	RF-GMV-07		
Traceability (backward)	Internal use case needs (not a specific scenario)		
Traceability (forward)	WP3-Requirements WP5-Requirements		
KPIs	K-U-06: Reduction of the use of computational resources.		

Updated version of the functional requirement RF-GMV-10

ID	RF-GMV-10	Priority	Must
Name	Robustness of the ODTS solution		
Description/ Rationale	The ODTS algorithm/ML model shall provide a navigation solution regardless of measurement gaps or communication failures.		

Dependency	RF-GMV-03
Traceability (backward)	Internal use case needs (not a specific scenario)
Traceability (forward)	WP3-Requirements WP5-Requirements
KPIs	K-U-05: Achievable distributed on-board ODTS performances versus the classical centralised on-ground ODTS.

In addition to the abovementioned modifications, a new set of requirements has been defined which establishes, in more detail, the needs for GMV's use case.

Firstly, the requirements RF-GMV-11 and RF-GMV-12 make clear GMV's intention to replace the orbital propagation and Kalman Filtering models for the realisation of the ODTS by ML/FL models. Also, requirements RF-GMV-13, RF-GMV-14 and RF-GMV-15 specify that these models, as well as the RL agents that can be introduced to address the scheduling problem, must be feasible in terms of memory and computational power for a possible implementation on representative on-board computing hardware.

New functional requirement RF-GMV-11

ID	RF-GMV-11	Priority	Must
Name	Implementation of an FL model for distributed ODTS		
Description/ Rationale	The distributed Extended Kalman Filter for the ODTS process developed for the baseline shall be replaced with a FL model		
Dependency	RF-GMV-01, RF-GMV-02		
Traceability (backward)	UC-03-Scenario 1 -		
Traceability (forward)	WP5-Requirements		
KPIs	K-U-05: Achievable distributed on-board ODTS performances versus the classical centralised on-ground ODTS. K-U-06: Reduction of the use of computational resources.		

New functional requirement RF-GMV-12

ID	RF-GMV-12	Priority	Should
Name	Substitution of the orbit propagator with a ML/FL model		
Description/ Rationale	The orbit propagator shall be replaced with a ML/FL model.		
Dependency	RF-GMV-07, RF-GMV-14		
Traceability (backward)	UC-03-Scenario 1 -		
Traceability (forward)	WP5-Requirements		
KPIs	K-U-06: Reduction of the use of computational resources.		

Funded by the European Union

New functional requirement RF-GMV-13

ID	RF-GMV-13	Priority	Should
Name	Feasibility of the ODTS ML model for on-board implementation		
Description/ Rationale	The ML model replacing the EKF shall fit into a representative satellite board in terms of memory and computation power.		
Dependency	RF-GMV-11, RF-GMV-20		
Traceability (backward)	UC-03-Scenario 1 -		
Traceability (forward)	WP5-Requirements		
KPIs	K-U-07: Software process development metrics based on ECSS standard. K-U-08: Software product metrics based on ECSS standard.		

New functional requirement RF-GMV-14

ID	RF-GMV-14	Priority	Should
Name	Feasibility of the orbit propagation ML/FL model for on-board implementation		
Description/ Rationale	The ML/FL model replacing the orbit propagator shall fit into a representative satellite board in terms of memory and computation power.		
Dependency	RF-GMV-12, RF-GMV-20		
Traceability (backward)	UC-03-Scenario 1 -		
Traceability (forward)	WP5-Requirements		
KPIs	K-U-07: Software process development metrics based on ECSS standard. K-U-08: Software product metrics based on ECSS standard.		

New functional requirement RF-GMV-15

ID	RF-GMV-15	Priority	Must
Name	Feasibility of the on-board implementation of the RL agents for ISL scheduling		
Description/ Rationale	The developed RL agents for the autonomous on-board ISL scheduling re-configuration functionality shall fit into a representative satellite board in terms of memory and computation power		
Dependency	RF-GMV-08, RF-GMV-20		
Traceability (backward)	UC-03-Scenario 2 -		
Traceability (forward)	WP5-Requirements		
KPIs	K-U-07: Software process development metrics based on ECSS standard. K-U-08: Software product metrics based on ECSS standard.		

Due to the updated direction in the progression of the GMV use case, with the growing interest in exploiting the optimization of the scheduling process of ISL connections, a set of

new requirements have also been added for this scenario regarding the links characterisation.

New functional requirement RF-GMV-16

ID	RF-GMV-16	Priority	Must
Name	Interconnection of all nodes on the constellation grid		
Description/ Rationale	The swarm network topology shall allow communications between any two nodes, at any moment in time, within maximum TBD timeslots		
Dependency	RF-GMV-08, RF-GMV-17, RF-GMV-18, RF-GMV-19		
Traceability (backward)	UC-03-Scenario 2 -		
Traceability (forward)	WP6-Requirements		
KPIs	N/A (Validated in WP7 demonstrations)		

New functional requirement RF-GMV-17

ID	RF-GMV-17	Priority	Must
Name	Linking capability of the nodes		
Description/ Rationale	Each satellite shall be capable of establishing links with up to two other satellites and a ground station simultaneously, provided that there is visibility among them.		
Dependency	RF-GMV-16, RF-GMV-18, RF-GMV-21		
Traceability (backward)	Internal use case needs (not a specific scenario)		
Traceability (forward)	WP6-Requirements		
KPIs	N/A (Validated in WP7 demonstrations)		

New functional requirement RF-GMV-18

ID	RF-GMV-18	Priority	Must
Name	Information accessibility between nodes		
Description/ Rationale	Each satellite, at every timeslot, shall be able to access the ephemerides of the satellites/ground stations it is supposed to connect with in the next timeslot in order to correctly point its antennas towards the target/s node/s.		
Dependency	RF-GMV-16, RF-GMV-17		
Traceability (backward)	UC-03-Scenario 1 -		
Traceability (forward)	WP6-Requirements		
KPIs	N/A (Validated in WP7 demonstrations)		

New functional requirement RF-GMV-19

ID	RF-GMV-19	Priority	Should
Name	Eventual consensus in scheduling re-configuration situation		

Description/ Rationale	The nodes of the swarm need to reach eventual consensus in case a failure occurs, and an autonomous on-board re-configuration of the nodes connections scheduling is required to happen, avoiding conflicts.
Dependency	RF-GMV-08, RF-GMV-16
Traceability (backward)	UC-03-Scenario 2 -
Traceability (forward)	WP4-Requirements
KPIs	N/A (Validated in WP7 demonstrations)

In order to develop the ML algorithms and simulate the entire swarm operation, a framework is needed. This shall allow the simulation of the communication among the nodes as well. Therefore, following the requirements RF-GMV-04 and RF-GMV-05, already defined in D2.2, additional requirements are provided.

New functional requirement RF-GMV-20

ID	RF-GMV-20	Priority	Must
Name	Testing framework		
Description/ Rationale	A framework for developing and testing the ML/FL/RL models performances shall be set-up.		
Dependency	RF-GMV-13, RF-GMV-14, RF-GMV-15, RF-GMV-21		
Traceability (backward)	Internal use case needs (not a specific scenario)		
Traceability (forward)	WP5-Requirements WP6-Requirements		
KPIs	N/A (Validated in WP7 demonstrations)		

New functional requirement RF-GMV-21

ID	RF-GMV-21	Priority	Must
Name	Replication of communication topology in the simulated network		
Description/ Rationale	The simulated network must be able to take into account the constraints related to the communication between the nodes of the swarm. For instance, a communication link between two nodes can only be established if they are visible to each other.		
Dependency	RF-GMV-17, RF-GMV-20		
Traceability (backward)	Internal use case needs (not a specific scenario)		
Traceability (forward)	WP6-Requirements		
KPIs	N/A (Validated in WP7 demonstrations)		

Finally, two more requirements have been added regarding both dynamic and static verification and correctness and safety ensurement of the final software models, which as was previously stated shall be feasible to be deployed on representative satellite boards.

Funded by the European Union

New functional requirement RF-GMV-22

ID	RF-GMV-22	Priority	Must
Name	Static and dynamic verification for on-board software		
Description/ Rationale	The models developed, which are intended to be feasible to be deployed on representative satellite boards, must be statically and dynamically verified.		
Dependency	No dependency with other requirements		
Traceability (backward)	Internal use case needs (not a specific scenario)		
Traceability (forward)	WP4 Requirements		
KPIs	K-U-07: Software process development metrics based on ECSS standard. K-U-08: Software product metrics based on ECSS standard.		

New functional requirement RF-GMV-23

ID	RF-GMV-23	Priority	Must
Name	Safety properties for on-board software		
Description/ Rationale	The code generation for the implementation on a representative board shall ensure correctness properties (i.e. preventing illegal memory accesses, ensuring loop termination, among others).		
Dependency	No dependency with other requirements		
Traceability (backward)	Internal use case needs (not a specific scenario)		
Traceability (forward)	WP4 Requirements		
KPIs	K-U-07: Software process development metrics based on ECSS standard. K-U-08: Software product metrics based on ECSS standard.		

3.2.4 Highly resilient factory shop floor digitalisation

This use case uses intelligent swarm systems on the factory shop floor to make it easier, more efficient, and less error-prone to use information technology in the automation of these highly collaborative processes that require the utmost levels of availability, resilience, and robustness. In this light, we add a requirement pertaining to the reaction to various error conditions (or similar) that can be easily customised to the needs of the factory management personnel, making operations more efficient by reducing reaction times in case of problems.

New functional requirement RF-ACT-24

ID	RF-ACT-24	Priority	Must
Name	Efficient and intuitive event trace pattern matching		
Description/ Rationale	Factory domain experts will define heuristics for various conditions in which certain reactions shall be triggered (be that alarms, mitigating actions, flagging events for further processing, etc.). These heuristics can be converted into monitoring code with a high degree of confidence regarding their correctness while being intuitive to read so that programmers and domain experts can both review them.		
Dependency	RF-ACT-06		

Traceability (backward)	UC-04-SC7 Logistics supervision
Traceability (forward)	N/A
KPIs	K-O-1.1

3.3 Federated learning orchestration verification requirements

In order to realise more reliable federated learning orchestration we have defined the requirements RNF-WP4-PROP-04 and RNF-WP4-VER-05 in D2.2. The first requirement collects a set of desirable properties that are specific to federated learning protocols, and the second establishes the need for realising a verification procedure to ensure these properties within the TaRDIS toolbox. The properties named in RNF-WP4-PROP-04 were collected in discussions with WP5 representatives, and these are also listed in D4.1. Up to now, the work towards fulfilling these requirements was naturally focused on the TaRDIS internal usage. We have started the preliminary investigation on verification of the FL orchestration focusing on the PTB-FLA tool (reported in this deliverable in Section 2.4.7). On the other hand, the use-case partners at the moment do not foresee these requirements as crucial for achieving their goals within the scope of TaDRIS.

Hence, the conclusion is that the requirements RNF-WP4-PROP-04 and RNF-WP4-VER-05 are desirable, but not necessary. For this reason, we have updated these requirements with respect to the priority level. As shown in the tables below, the priority has now been set to “Could” instead of “Must” that was indicated previously.

Updated version of the requirement RNF-WP4-PROP-04

ID	RNF-WP4-PROP-04	Priority	Could
Name	WP4 - Properties - Decentralised Machine Learning Models		
Description/ Rationale	Description: The TaRDIS models must support desirable federated learning (FL) properties, including FL roles of agents, FL data privacy, FL message delivery, and FL clients equality. Rationale: Identifying and expressing properties for correct workflow orchestration in FL algorithms, including FL restricted resource usage on edge devices with different capabilities and/or roles.		
Dependency	RF-WP5-FL-ALG-05		
Traceability (backward)	Deployment and orchestration integration in WP4 (D4.1) RF-TID-06 (FL workflow orchestration) RF-ACT-14 (FL restricted resource usage) RF-GMV-02 (FL restricted resource usage) RF-WP3-MOD-02 (Verification of application correctness) RF-WP3-MOD-05 (Specifying device capabilities) RF-WP5-FL-ALG-01 (Implemented FL algorithms) WP5-RF-RLALG-2 (Centralised AI orchestration) WP5-RF-RLALG-3 (Decentralised AI orchestration) RNF-WP3-GEN-02 (Verification of correctness)		

Traceability (forward)	RNF-WP4-VER-05 (Verification - FL Orchestration) Deployment and orchestration integration in WP4 (D4.2, D4.3)
KPIs	K-O-2: development environment and verification K-O-3: decentralised intelligence

Updated version of the requirement RNF-WP4-VER-05

ID	RNF-WP4-VER-05	Priority	Could
Name	WP4 - Verification - Federated Learning Orchestration		
Description/ Rationale	Description: The TaRDIS toolbox must provide facilities to verify federated learning orchestration. Rationale: Verification and validation of identified FL properties based on CSP calculus for modelling and PAT model checker for verification, which will be integrated with Multiparty Session Types newly developed techniques.		
Dependency	RNF-WP4-PROP-04 (Properties - decentralised ML models)		
Traceability (backward)	Deployment and orchestration integration in WP4 (D4.1) RNF-WP4-VER-01 (Verification - communications behaviour) RNF-WP4-PROP-04 (Properties - decentralised ML models) RF-TID-06 (FL workflow orchestration) RF-ACT-14 (FL restricted resource usage) RF-GMV-02 (FL restricted resource usage) RF-WP3-MOD-02 (Verification of application correctness) RF-WP5-FL-ALG-01 (Implemented FL algorithms) WP5-RF-RLALG-2 (Centralised AI orchestration) WP5-RF-RLALG-3 (Decentralised AI orchestration) RNF-WP3-GEN-02 (Verification of correctness)		
Traceability (forward)	Deployment and orchestration integration in WP4 (D4.2, D4.3)		
KPIs	K-O-2: development environment and verification K-O-3: decentralised intelligence		

3.4 Updates on Toolbox Requirements

This section lists requirements that are not included in the rest of the document and details how we will proceed with them. This can either be removal, modification, skipping (within this report), or deferral for the specification of details that are yet unknown.

Currently not covered requirements (sans end-user requirements from UC)

requirement ID and name	**action**	**comment**
RF-WP2-GEN-01 IDE	skip	#1
RF-WP2-GEN-02 IDE integration	skip	#1

requirement ID and name	action	comment
RF-WP2-GEN-03 IDE support	skip	#1
RF-WP2-GEN-04 IDE support to TaRDIS	skip	#1
RF-WP3-MOD-05 graphical representation	remove	this is a duplicate (i.e., error in D2.2)
RF-WP3-GEN-06 interfacing with pre-existing middleware and services	defer	#2
RNF-WP4-PROP-02 data management and replication	modify	#3
RNF-WP4-VER-02 distributed data management	modify	#3

comments:

#1 The IDE requirements are not reported upon as part of the toolbox specification because of the dedicated IDE reports D3.4 and D3.6 that are scheduled for the remainder of the project.

#2 The ability of TaRDIS applications to interface with external services or middlewares is crucial for industrial use case realisation, but none of the tools themselves implement this part—this is rather a requirement in terms of not making it inconvenient (or even impossible) for the developer to compose these aspects within a single application.

#3 Since the use cases themselves have no direct requirement on the formal properties and their verification for data management and replication, we use these requirements only internally and change their priority from “Must” to “Could”.

Funded by the European Union

4 Conclusions

In this third and last report of work package 2 we have concluded our definition of heterogeneous swarm systems by specifying the TaRDIS toolbox with which application developers will be well-equipped for implementing challenging use cases in this domain. We have categorised our tools into the various purposes and activities that a programmer will need to perform when developing a swarm application, covering the aspects that are contributed by the four technical work packages:

- programming abstractions and overall approach (WP3)
- program logic and analysis (WP4)
- decentralised machine learning (WP5)
- data management and distribution (WP6)

While some tools internally build upon other TaRDIS tools, the whole toolbox is available to application developers so that they can freely compose their applications using the right tool for each job.

While matching each TaRDIS tool that has been, is, or will be developed in the scope of the project to its purpose's set of defined requirements, we have found that the projected toolbox matches the overall use case requirements quite well. We reported on these in deliverable D2.2 earlier this year. Some small mistakes in that report have been identified and errata reported in section 3 above. We have also found that some technical requirements projected earlier do not in have counterparts in the industrial use cases, in which cases we have removed them or downgraded to "could" priority—these requirements may still serve to guide the precise implementation choices of TaRDIS tools within the second half of the project.

With this report we now have comprehensive documentation on the tools that will be available for the final implementation of the industrial use cases, which is the basis for the upcoming report D7.2 that will also contain the criteria for assessing the TaRDIS toolbox at the end of the project.

Funded by the European Union