D3.1: First Report on Programming Model and APIs

Revision: **1.1**

Work package	WP3
Task	T3.1, T3.2
Due date	31/Dec/2023
Submission date	9/Feb/2024
Deliverable lead	Alceste Scalas (DTU)
Version	**1.1**
Authors	João Costa Seco (NOVA), Cláudia Soares (NOVA), Carla Ferreira (NOVA), João Leitão (NOVA), António Ravara (NOVA) Nicolas Kourtellis (TID), Dimitra Tsigkari (TID) Carlos Reis (CMS), Carlos Coutinho (CMS) Giovanni Granato (GMV) Sebastian Alexander Mödersheim (DTU) Ping Hou (OXF), Nobuko Yoshida (OXF) Rafael Oliveira Rodrigues (EDP CNET), Manuel Pio Silva (EDP CNET) Dušan Jakovetić (UNS), Lidija Fodor (UNS), Miroslav Zarić (UNS), Miroslav Popovic (UNS) Sotirios Spantideas (NKUA) Roland Kuhn (ACT)
Internal Reviewers	Ana Ribeiro (NOVA) Amrita Prasad (Martel)
Abstract	This document reports the initial design of the programming model and APIs which will be offered by the TaRDIS toolbox. The model and APIs are illustrated using selected elements of the TaRDIS use case applications as a reference.
Keywords	decentralised programming toolbox, models, APIs

www.project-tardis.eu

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

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

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

© 2023 - 2025 TaRDIS Consortium

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 aims at building a distributed programming toolbox to simplify the development of decentralized, heterogeneous swarm applications deployed in diverse settings. This report documents the current status in the design and development of the TaRDIS programming model and APIs.

The main contribution of this deliverable is the first version of the TaRDIS programming model and the first outline of the TaRDIS toolkit APIs. The deliverable explains how the TaRDIS toolbox will support two main kinds of swarm components — called *internal services* and *perimeter services*. To describe the intended use of the TaRDIS programming model and APIs, this deliverable also outlines the APIs that will be made available by each work package (as part of the TaRDIS toolbox), and how each use case plans to leverage them. This contribution is a crucial stepping stone towards the project objectives, ensuring the alignment of the various work packages.

Funded by the European Union

© 2023-2025 TaRDIS Consortium

Table of Contents

1 INTRODUCTION 8
2 PROGRAMMING MODEL OVERVIEW AND DESIGN METHODOLOGY 9
2.1. The TaRDIS Programming Model and Toolbox Design 9
2.1.1. Internal TaRDIS Services, Swarm Protocols, and Workflows 10
2.1.2. Perimeter TaRDIS Services 12
2.1.3. Internal vs. Perimeter TaRDIS Services: Trade-Offs 13
2.2. TaRDIS Programming Model Extensions via DCR Graphs 13
2.2.1. An Overview of ReGraDa / DCR Graphs 14
2.3. How the TaRDIS Use Cases Will Leverage the TaRDIS Toolbox 16
2.4. Cross-Language Interoperability 17
3 OVERVIEW OF THE TaRDIS APIS 19
3.1. Initial API Sketches 19
3.1.1 Event-Driven API for Internal Services: Instantiating a TaRDIS Swarm 20
3.1.2 Event-Based Input-Output API for Perimeter Services 21
3.1.3 Machine Learning APIs 24
3.2. Analysis and Verification Facilities 24
3.2.1. Specifying and Verifying Communication Behaviour - T4.1 24
3.2.2. Specifying and Analysing Data Consistency - T4.2 27
3.2.3. Specifying and Analysing Security Properties - T4.3 29
3.2.4. Deployment and Orchestration Integration - T4.4 32
3.3. Artificial Intelligence and Machine Learning APIs 32
3.3.1. AI/ML Programming Primitives - T5.1 32
3.3.2. AI-Driven Planning, Deployment, and Orchestration - T5.2 34
3.3.3. Lightweight and Energy-Efficient ML Techniques - T5.3 43
3.4. Data Management and Distribution Primitives 44
3.4.1. Decentralised Membership and Communication APIs - T6.1 44
3.4.2. Decentralised Data Management and Replication APIs - T6.2 49
3.4.3. Decentralised Monitoring and Reconfiguration APIs - T6.3 51
4 OVERVIEW OF THE TaRDIS USE CASE APPLICATIONS 52
4.1. Actyx 52
4.1.1. Actyx App 0: Machine Requesting Maintenance 52
4.1.2. Actyx App 1: Maintenance Worker Tablet 53
4.1.3. Actyx App 2: Manager Dashboard 56
4.1.4. Actyx App 3: Real-Time Monitoring 58
4.2. EDP 58
4.2.1. Background and general objective of the Energy use case 58
4.2.2. Energy use case components and objectives 58
4.2.3. Working Principles 59
4.2.4. Requirements to TaRDIS 59
4.2.5. Scenarios 59
4.3. GMV 68

© 2023-2025 TaRDIS Consortium

4.3.1. Sequential (Centralised) Orbit Simulation Application 68
4.3.2. Distributed Simulation Application Based on PTB-FLA 69
4.3.3. Roadmap 69
4.4. Telefónica 69
4.4.1. Analyses for security (T4.3, Section 3.2.3) 70
4.4.2. AI/ML programming primitives (T5.1, Section 3.3) 70
4.4.3. Lightweight and energy-efficient ML library (T5.3, Section 3.3.3) 71
4.4.4. Decentralised membership and communication (T6.1, Section 3.4.1) 71
4.4.5. Decentralised monitoring and reconfiguration (T6.3) 72
5 CONCLUSION 73

Funded by the European Union

© 2023-2025 TaRDIS Consortium

ABBREVIATIONS

API	Application Programming Interface
AGV	Automated Guided Vehicle
BDS-3	BeiDou 3rd Generation navigation satellite system
CDF	Cumulative Distribution Function
DCR	Dynamic Condition Relation
DER	Distributed Energy Resources
DL	Deep Learning
DNN	Deep Neural Network
DP	Differential Privacy
DRFL	Deep Reinforcement Federated Learning
DSO	Distribution System Operator
ERP	Enterprise Resource Planning
FL	Federated Learning
FLaaS	Federated Learning as a Service
G2G	Galileo 2nd Generation of satellites
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
ISL	Inter-Satellite-Link
IP	Internet Protocol
IPFS	InterPlanetary File System
JS	JavaScript
LEO	Low Earth Orbit
LSTM	Long Short-Term Memory
MES	Manufacturing Execution System
ML	Machine Learning
MPST	Multiparty Session Types
ODTS	Orbit Determination and Time Synchronization

Funded by the European Union

© 2023-2025 TaRDIS Consortium

P2P	Peer-to-Peer
PNT	Position, Navigation and Timing
SGAM	Smart-Grid Architectural Model
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Funded by the European Union

© 2023-2025 TaRDIS Consortium

1 INTRODUCTION

This report documents the ongoing work on the design and development of the TaRDIS programming toolkit — specifically, its programming models and APIs.

The TaRDIS programming model and APIs are central aspects of the project that require a clear alignment between the requirements of the use cases, and the outputs of the research and development work packages. Consequently, the definition and development of the programming model and APIs is a collaborative effort that requires a close collaboration among all project partners.

This document has been developed concurrently with Deliverable D2.2 ("Report on overall requirements analysis"), and the two deliverables complement each other: specifically, the use case excerpts described in this document are not exhaustive, and they focus on outlining the intended use of key aspects of the TaRDIS APIs. The alignment between the TaRDIS model and APIs (produced by WP3) and the use case and toolbox requirements (produced by WP2) will be achieved throughout the rest of the project.

This document has the following structure:

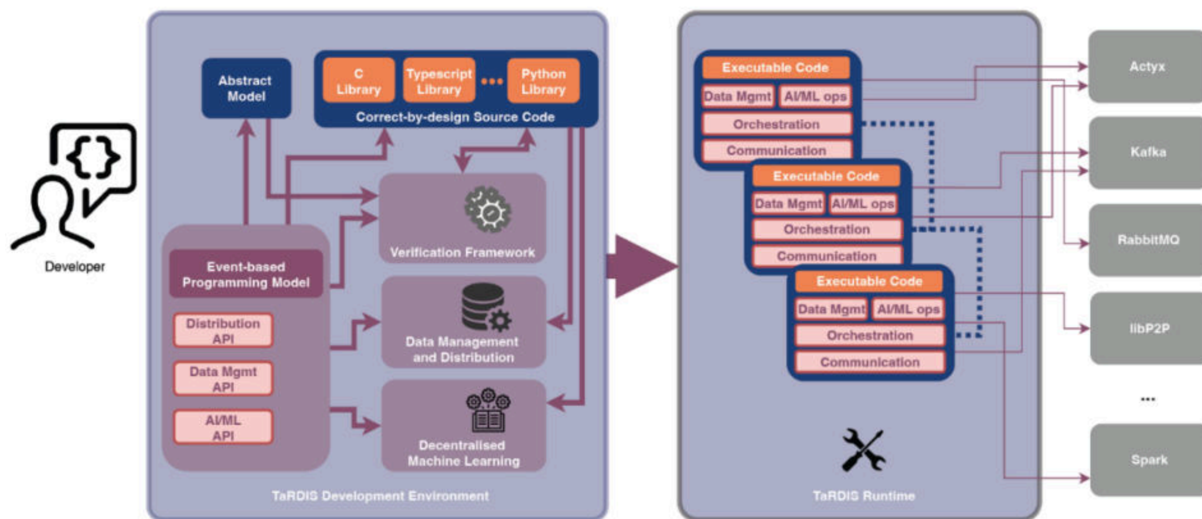
- Section 2 outlines the TaRDIS programming model, and the methodology and considerations leading to its design. It also provides a table that summarises how each use case plans to leverage the various facets of the TaRDIS toolbox.
- Section 3 provides an overview of the APIs that will be offered by the TaRDIS toolbox.
- Section 4 provides an overview of relevant parts of the project use cases, with the purpose of illustrating how each use case plans to use the TaRDIS model and APIs.

The conclusion (Section 5) summarises the main outcomes and outlines the next steps.

© 2023-2025 TaRDIS Consortium

2 PROGRAMMING MODEL OVERVIEW AND DESIGN METHODOLOGY

The TaRDIS project proposal envisions an event-driven programming model and toolbox allowing application programmers to take advantage of various facilities (communication, verification, machine learning, monitoring and reconfiguration…) helping them develop safe and reliable distributed swarm applications. Such facilities are made available through the "TaRDIS Runtime" — which provides various higher-level APIs and abstractions over lower-level libraries and services; moreover, the proposal envisions dedicated IDE support to simplify the programmers' tasks. This vision is summarised in the figure below (from the TaRDIS project proposal); here the "abstract model" is an abstract representation of a TaRDIS application, which enables the use of the toolbox verification facilities.



This vision has been refined and made more concrete during the first months of the TaRDIS project. Since June 2023, WP3 has organised a series of **7 Workshops on Models and APIs** where all project partners have collaborated in analysing and discussing the definition of the TaRDIS model and APIs: this close interaction ensures the alignment between the ongoing work and outputs of the research work packages (WP4, WP5, WP6) and the requirements and suggestions from the industry partners (stemming from WP2 and WP7). The present deliverable is one of the outcomes of this collaborative process.

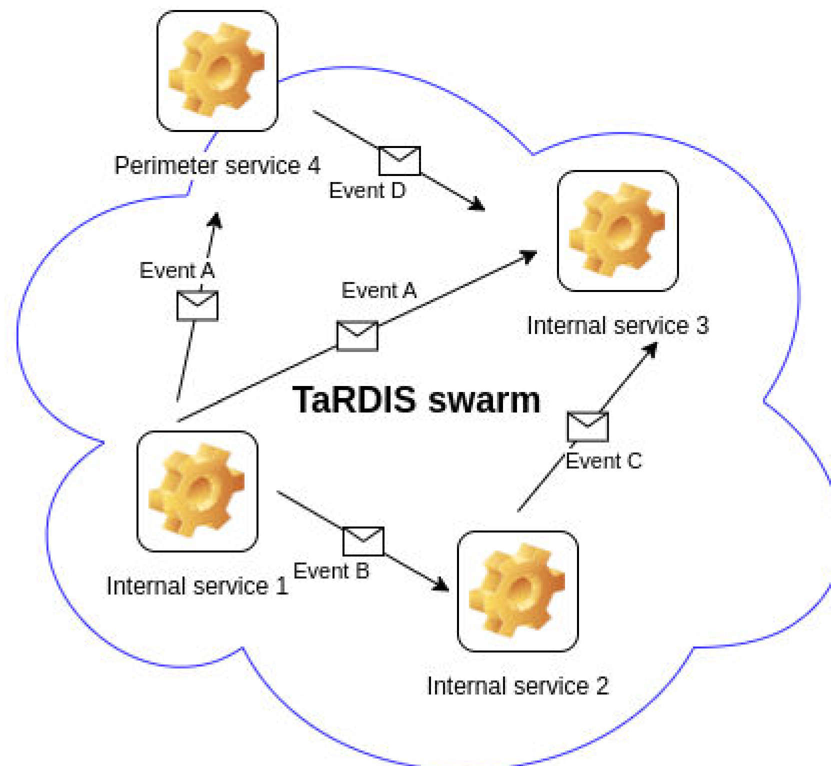
The rest of this section presents an overview of the design of the TaRDIS programming model and toolbox (Section 2.1), and discusses a possible extension based on DCR graphs (Section 2.2). Then, it outlines how each use case plans to leverage the TaRDIS toolbox (model and APIs, Section 2.3), and how cross-language interoperability will be addressed (Section 2.4).

2.1. The TaRDIS Programming Model and Toolbox Design

Given the variety of the use cases and their requirements, the TaRDIS programming model is being designed to support the development of **swarm applications** combining two main kinds of programs: **internal TaRDIS services** and **perimeter TaRDIS services**. The intuition is the following (and is depicted in the figure below):

© 2023-2025 TaRDIS Consortium

- a **TaRDIS swarm application** is an ensemble of concurrent, distributed, and possibly heterogeneous **services** which interact over a network in an event-driven fashion, using the communication facilities provided by the TaRDIS toolbox;
- an **internal service** can be more deeply integrated in a TaRDIS swarm application. We foresee that internal services will have more complete access to the TaRDIS toolbox - in particular, to its higher-level APIs, and its verification capabilities;
- a **perimeter service** can communicate with a TaRDIS swarm application and use (most of) the TaRDIS APIs. However, a perimeter service may not have complete access to the TaRDIS toolbox - in particular, it might not take advantage of its verification capabilities, and may have limited access to its higher-level APIs.



The following subsections illustrate in more detail the differences between internal services (Section 2.2.1) and perimeter services (Section 2.1.2), and their respective trade-offs (Section 2.1.3).

2.1.1. Internal TaRDIS Services, Swarm Protocols, and Workflows

An **internal TaRDIS service** is a program that does not directly control its main execution loop: instead, the program is written as a series of reactive call-back functions that are executed by the **TaRDIS execution engine** depending on preconfigured **events** (e.g. the arrival of a certain type of message in a certain state of the application).

Internal TaRDIS services follow the **TaRDIS workflow model**, i.e. they can be intuitively depicted as state machines where state transitions are triggered by events, and call-back functions are executed when entering or leaving a state. A workflow describes the behaviour of an individual service that joins a larger swarm. The state transitions, event dispatch, and call-back execution described in the workflow are handled by the aforementioned TaRDIS execution engine.

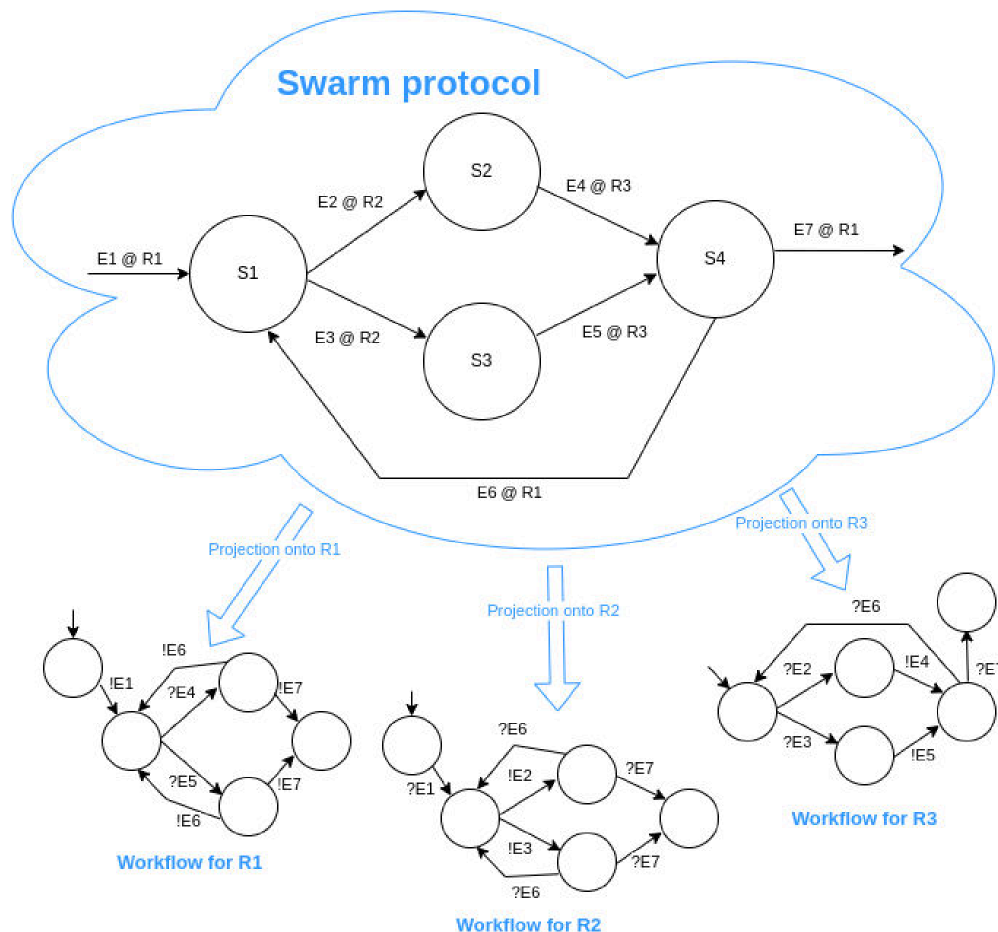
© 2023-2025 TaRDIS Consortium

Related kinds of event-driven programming styles and reactive execution engines are offered by well-established distributed application programming libraries like Twisted,[1] Akka,[2] or Orleans[3] — but the TaRDIS workflow model will be able to describe more complex applications with *stateful* communication protocols. Moreover, TaRDIS will offer tooling to ensure that a program actually respects a desired workflow.

Developers are free to deploy services that follow arbitrary workflows and make them join a same swarm application — however, if the individual workflows are incompatible with each other, the overall application may misbehave (e.g. resulting in deadlocks, or communication of events that are not handled by the intended recipients). To address this issue, TaRDIS plans to provide:

- the possibility of specifying **swarm protocols**, which provide a global bird's eye view of the intended behaviour of all components that might join a swarm application (each one implementing a specific **role**) to produce and consume **events**; and
- tools to **project (i.e., synthesise) a local workflow out of a swarm protocol**, ensuring that the local behaviour of a service is compatible with the rest of the swarm.

The idea is illustrated in the figure below:



In the figure above:

- The swarm protocol describes a "global distributed state machines" where the edges correspond to the intended interactions between roles R1, R2, and R3; such roles, in

[1] https://twisted.org/
[2] https://akka.io/
[3] https://github.com/dotnet/orleans

© 2023-2025 TaRDIS Consortium

turn, may produce and consume events E1..E7 (the notation "E @ R" means that event E is produced by some swarm participant having role R). These events advance the overall state of the protocol.
- The swarm protocol is projected into workflows for the roles R1, R2, and R3: for instance, the workflow for role R2 says that a swarm participant implementing role R2 is expected to await event E1, and then emit one of the events E2 or E3, and then await E6 (looping back to a previous state) or E7 (which terminates the workflow).

The general idea of projecting correct-by-construction local specifications (called "workflows" in TaRDIS) from a global protocol specification (called "swarm protocol" in TaRDIS) can be found e.g. in the literature on cryptographic protocols (as "Alice-and-Bob notation"[4] or "protocol narrations"[5]) and modelling and verification of concurrent and distributed systems (e.g. multiparty session types[6]). The availability of the global swarm protocol specification has two advantages:

1. it provides an intuitive overview of the system behaviour that can be easier to understand by non-experts, and
2. enables better analysis of the system behaviour using the analysis methodologies and tools developed in WP4 (see the TaRDIS Deliverable D4.1).

Concretely, the TaRDIS swarm protocol, workflow model, and execution engine are being designed and developed using as a starting point the *machine runner*[7] model and tooling created and released (under Open Source license) by the project partner Actyx: their approach is described in recent publications[8] [9] and is being discussed and adapted and as part of WP3, analysed as part of WP4, and implemented and improved as part of WP6 and WP7. Examples of swarm protocol and workflow applications can be found in the "Actyx" section of the use cases overview (Section 4.1).

In addition to the "top-down" approach outlined above, the TaRDIS project may also explore novel "bottom-up" development methods and tools to directly check whether different workflows can be correctly composed, even if a swarm protocol is not provided beforehand.

2.1.2. Perimeter TaRDIS Services

A **perimeter TaRDIS service** is a program that does *not* delegate its main execution loop to the TaRDIS execution engine, hence does *not* follow the TaRDIS swarm protocol/workflow model outlined in Section 2.1.1. A TaRDIS perimeter service might directly control its main execution loop (or delegate it to other libraries, e.g. GUI toolkits like Qt[10]), and may use only selected (and typically lower-level) TaRDIS APIs for specific purposes - e.g. producing or

[4] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21, 2 (Feb. 1978), 120–126. https://doi.org/10.1145/359340.359342
[5] Sébastien Briais, Uwe Nestmann: A Formal Semantics for Protocol Narrations. Trustworthy Global Computing 2005. Lecture Notes in Computer Science. Vol. 3705. pp. 163–181. https://doi.org/10.1007%2F11580850_10
[6] Kohei Honda, Nobuko Yoshida, Marco Carbone: Multiparty Asynchronous Session Types. J. ACM 63(1): 9:1-9:67 (2016). https://doi.org/10.1145/2827695
[7] https://www.npmjs.com/package/@actyx/machine-runner
[8] Roland Kuhn, Hernán C. Melgratti, Emilio Tuosto: Behavioural Types for Local-First Software. ECOOP 2023: 15:1-15:28. https://doi.org/10.4230/LIPIcs.ECOOP.2023.15
[9] Roland Kuhn, Alan Darmasaputra: Behaviorally Typed State Machines in TypeScript for Heterogeneous Swarms. ISSTA 2023: 1475-1478. https://doi.org/10.1145/3597926.3604917 - https://doi.org/10.48550/arXiv.2306.09068
[10] https://www.qt.io/product/framework

Funded by the European Union

© 2023-2025 TaRDIS Consortium

awaiting some events, accessing communication or AI/ML primitives, etc. Generally speaking, a perimeter TaRDIS service will use the lower-level APIs provided by the TaRDIS toolbox.

2.1.3. Internal vs. Perimeter TaRDIS Services: Trade-Offs

The main advantage of writing an **internal TaRDIS service** is that the programmer has more complete access to the verification capabilities of WP4 (ensuring e.g. that the code follows a desired communication protocol, or handles its data in a correct way); moreover, the programmer can access higher-level APIs that can hide more details about communication and distribution. This is made possible by the fact that an internal service has to follow the TaRDIS workflow model, hence the TaRDIS verification tools and the toolbox itself can make more assumptions about the service's behaviour. The main drawbacks in developing an internal TaRDIS service are:

- a steeper learning curve, as the programmer needs to become familiar with the TaRDIS workflow model, the execution engine, and the resulting programming style; and
- the need to delegate the main execution loop to the TaRDIS execution engine, which may make it harder to integrate already-existing applications written in a different style.

To avoid these drawbacks, a programmer may choose to write a **perimeter TaRDIS service** instead. In this case, the entry barrier is quite low: a programmer only needs to start using the relevant parts of the TaRDIS APIs (e.g. for communication or AI/ML). This smooths out the learning curve and simplifies the task of adapting an already-existing application to interact with a TaRDIS swarm. The main drawbacks in developing a perimeter TaRDIS services are:

- the programmer may not have access to all APIs provided by the TaRDIS toolbox. In general, higher-level APIs may not be usable, and the programmer may need to use the lower-level APIs, which may be more verbose and easier to misuse; moreover,
- the programmer cannot rely on the verification capabilities developed in WP4 (because a perimeter TaRDIS service can be written in any style and can be very dynamic, thus going beyond the current capabilities of software verification).

These drawbacks may increase the risk of writing more code and introducing bugs.

By developing a programming model that supports both internal and perimeter services, we aim at maximising the adoption of the TaRDIS toolbox: programmers may start writing perimeter services (possibly by adapting already-existing programs), and progressively consider the introduction of internal services as their applications evolve and the cost-benefit trade-off becomes clearer.

2.2. TaRDIS Programming Model Extensions via DCR Graphs

The definition of internal services in a TaRDIS swarm application (Section 2.1.1), where the workflow model is central to the behaviour of the swarm, demands specification and programming languages that are both flexible, and human readable. In this section, we introduce an extension of the language ReGraDa (Dynamic Condition Relation (DCR)

© 2023-2025 TaRDIS Consortium

Graphs, with reactive computation and data mapped to a graph database)[11] [12] that provides a business process-like language to specify and evolve the behaviour of a swarm. We are extending our implementation to incorporate choreographies in the style of Hildebrandt's seminal work[13] [14]. This extension is being actively researched (as part of WP3 and WP4) and evaluated for inclusion in the TaRDIS toolkit.

Prior work exists to express a variety of workflow features in a declarative, stateful, and flexible way, notably ranging from relationships between activities, dynamic creation of behaviour, time constraints, and the relation with data and computation. However, the current state of the art requires a centralized or synchronized management of the workflow state.

General DCR graphs are strictly more expressive than the state machines in the TaRDIS workflow model and therefore this extension needs to be carefully studied, and fragments of the language can be adopted to maintain the guarantees of the graph in the underlying infrastructure. As a first step we plan to study the distribution and execution of choreographies in swarms and study the consistency of the state of the workflow in a decentralized environment. For instance, if there are causal dependencies between events in the swarm that are executed by independent agents, how can we guarantee it without a centralized entity?

Prior work shows that DCR graphs can be mapped to a state-machine formulation.[15] In particular, events in the workload are translated into events that the state-machine can asynchronously process, and the enableness of an event (a central property to allow the execution/triggering of an event) can be mapped directly to the existence of a given transition on a particular state (or set of states). Our plan is to handle events across different machines or agents (each one exciting its own state machine) taking advantage of decentralised communication and storage mechanisms developed in the context of WP5. This allows, among other things, to use these primitives to disseminate or route events generated by one local workflow to all other workflows that might need it, or to store these events on decentralised storage solutions to enabled the processing of events in a asynchronous way when a state machine that should process that event is offline or unreachable at the time of the event emission. Formally specifying how to model the conversion of the DCR graph model towards a state-machine based implementation (or skeleton) is going to be a topic of research in TaRDIS in the context of WP3.

[11] Galrinho, L., Seco, J.C., Debois, S., Hildebrandt, T., Norman, H., Slaats, T. (2021). ReGraDa: Reactive Graph Data. In: Damiani, F., Dardha, O. (eds) Coordination Models and Languages. COORDINATION 2021. Lecture Notes in Computer Science(), vol 12717. Springer, Cham. https://doi.org/10.1007/978-3-030-78142-2_12

[12] Eduardo Geraldo, João Costa Seco, and Thomas Hildebrandt. 2023. Data-Dependent Confidentiality in DCR Graphs. In Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming (PPDP '23). Association for Computing Machinery, New York, NY, USA, Article 7, 1–13. https://doi.org/10.1145/3610612.3610619

[13] Thomas T. Hildebrandt, Hugo A. López, Tijs Slaats: Declarative Choreographies with Time and Data. BPM (Forum) 2023: 73-89

[14] Thomas T. Hildebrandt, Tijs Slaats, Hugo A. López, Søren Debois, Marco Carbone: Declarative Choreographies and Liveness. FORTE 2019: 129-147

[15] Thomas T. Hildebrandt, Raghava Rao Mukkamala. 2010. Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In the Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010: 59-73.

Funded by the European Union

© 2023-2025 TaRDIS Consortium

2.2.1. An Overview of ReGraDa / DCR Graphs

We now briefly describe ReGraDa / DCR graphs. A graph, representing a workflow, includes input data elements that act as events triggered by an external agent and represent user/system interactions, and data elements or computation events that represent internal events and computations. These data elements also work as persistent records in a database or distributed storage system.

We use the textual syntax in this section, which is interchangeable with the visual notation that can be found in the literature.

The syntax for input events is as follows:

```
x:Label[?: Type]: R1 -> R2
```

Each event declaration has an identifier (x), visible in its definition scope, a label (Label) that identifies the type of events, it features a question mark to signify that it defines an input event, and a type to characterize the type of values that it receives. Events in this abstraction represent messages exchanged by the agents in the system. The declaration of an event also includes two roles: one for the emitter of the message and another for the receptor of the message.

Data elements, or computation events are defined as follows:

```
x:Label[Expression]: R1 -> R2
```

All events are assigned a state consisting of three Boolean values and one value of a given type. The Boolean values indicate respectively if the event was executed (or happened) previously, if it is included, i.e. visible and ready to be executed (or to happen), and if the event is pending response, this means that all pending events must be executed for the process to be considered in a terminal state. When assigning roles to events, we define a choreography, where the execution of an event corresponds to a message being sent between two (or more) members of a choreography (R1 and R2 above). The "happened" state of an event is set by executing an event, and the pending response state is set to false by executing an event. The "included" state is changed by the control-flow relations and the execution of the preceding events in the graph. The value associated with an event is obtained by external sources in input events and computed internally in other cases by evaluating the associated expression. The state is kept by a mapping from events to state, and the initial value is "not happened", "included", and "not pending". The symbols "%" and "!" can be used in declarations to represent that the initial state is "excluded" and "pending" respectively.

The structure of a process P is defined by a sequence of event declarations and a sequence of control flow and data relations (for simplicity we omit data relations in the following graphs).

The control flow relations that are used are the following:

- Condition relation (`A -->* B`) means that event B cannot happen before event A
- Response relation (`A *--> B`) means that if A happens, B becomes a pending response event (must be executed in the future)
- Milestone relation (`A --><> B`) means that if A is pending, B cannot be executed.
- Include relation (`A -->+ B`) means that if A happens, B becomes included in the process.

© 2023-2025 TaRDIS Consortium

- Exclude relation (`A -->% B`) means that if A happens, B becomes included in the process.
- Spawn relation (`A -->> P`) means that if A happens, the elements of process P are instantiated and added to the process.

Note: we are still not considering timeouts in this example, but those will be necessary to model certain parts of the scenarios.

For example, consider the global workflow (with simplified syntax ignoring data and roles):

```
  r:request
% s:start
% f:finish
;
r -->+ s
r *--> s
r -->% r
s -->+ f
s *--> f
s -->% s
f -->% f
```

This workflow has three events to request a task to be made, to start the task and finish the task. Once the event `r` (request) is executed the event `s` (start) is included in the workflow and made pending response (the `-->+` and `*-->` relations between `r` and `s`). The event `r` excludes itself (the `-->%` relation), and can be executed only once. A similar behaviour is defined for the start and finish events that need to be executed in sequence and only once for the workflow to reach a final state.

In the description of the EDP use case (Section 4.2), the reader may find examples of DCR/ReGraDa graphs to illustrate the application of this extension in the context of TaRDIS.

2.3. How the TaRDIS Use Cases Will Leverage the TaRDIS Toolbox

The following table summarises how each use case of the TaRDIS project plans to leverage the TaRDIS toolbox. Each column and row header in the table links to the corresponding subsection of the API overview (Section 3) and the use case applications overview (Section 4).

	Actyx Application 1 (Section 4.1.2)	Actyx Application 2 (Section 4.1.3)	EDP (Section 4.2)	GMV (Section 4.3)	Telefónica (Section 4.4)
Service type(s) (Section 2.1)	1+ internal (100x) 1 perimeter	1 internal (1000x) 1 perimeter	Internal	Perimeter	Perimeter
Programming language(s)	TypeScript	TypeScript	Java/Python	Matlab/ Python	Python/Java

© 2023-2025 TaRDIS Consortium

	Actyx Application 1 (Section 4.1.2)	Actyx Application 2 (Section 4.1.3)	EDP (Section 4.2)	GMV (Section 4.3)	Telefónica (Section 4.4)
Specifying and analysing communications behaviour (T4.1, Section 3.2.1)	✅				
Specifying and analysing data consistency (T4.2, Section 3.2.2)					
Specifying and analysing security properties (T4.3, Section 3.2.3)					
Deployment & orchestration integration (T4.4, Section 3.2.4)					
AI/ML programming primitives (T5.1, Section 3.3.1)					
AI-driven planning, deployment & orchestration (T5.2, Section 3.3.2)					
Lightweight and energy-efficient ML library (T5.3, Section 3.3.3)					
Decentralised membership and communication (T6.1, Section 3.4.1)					
Decentralised data management and replication (T6.2, Section 3.4.2)					
Decentralised monitoring and reconfiguration (T6.3, Section 3.4.3)					✅

Funded by the European Union

© 2023-2025 TaRDIS Consortium

2.4. Cross-Language Interoperability

The TaRDIS toolkit aims at being language-independent and offering APIs that can be leveraged by multiple programming languages; this is necessary for the development of heterogeneous distributed applications where different services may be implemented using different programming languages.

At this stage, the APIs and facilities provided by WP4, WP5, and WP6 are being prototyped and developed using the most suitable (given their applications) programming languages — but the plan is to make such APIs available to other languages, too (giving a higher priority to the programming languages used by the TaRDIS use case applications). To achieve this, we will proceed in two phases:

- As a first step, we plan to leverage cross-language translation layers to make an API developed with programming language A available to programs written in programming language B ≠ A. To this end, we may e.g. expose the APIs via gRPC,[16] OpenAPI,[17] or WebSockets[18]/JSON schema.[19] When possible, more direct integrations will be evaluated — e.g. APIs written in C, C++, Rust, or Go can be often directly used from Python code through its `ctypes` module,[20] and from JVM-based languages through the Foreign Function and Memory API.[21] This integration will be sufficient for developing application prototypes and inform the next step.

- In a second phase, the integration will be refined based on the findings in the first step (with improvements e.g. in terms of performance or usability when needed).

[16] https://grpc.io/
[17] https://www.openapis.org/
[18] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[19] https://json-schema.org/
[20] https://docs.python.org/3/library/ctypes.html
[21] https://docs.oracle.com/en/java/javase/21/core/foreign-function-and-memory-api.html

Funded by the European Union

© 2023-2025 TaRDIS Consortium

3 OVERVIEW OF THE TaRDIS APIS

This section outlines the ongoing work in the design and development of the TaRDIS APIs. Here we use the term "API" in a broad sense, covering all facilities that will be made available to programmers that use the TaRDIS toolkit; this includes both APIs in the "classic" sense (i.e. the specification of functions, procedures, and methods callable by user's code), and facilities and tooling supporting the programmer (e.g. the verification facilities developed in WP4).

The contents of this section represents the current status and plans for the API work, which will evolve throughout the rest of the TaRDIS project. The titles of each subsection highlight which WP is working on the related APIs and features of the TaRDIS toolbox.

- Section 3.1 provides some initial API sketches, outlining how some key facilities of the TaRDIS toolbox will be organised and made available to programmers.
- Section 3.2 outlines the analysis and verification facilities.
- Section 3.3 outlines the APIs that will be provided for AI and machine learning-related functionality.
- Section 3.4 outlines the data management and distribution primitives.

3.1. Initial API Sketches

This section provides initial sketches of the TaRDIS APIs, organised by functionality. These sketches are not complete: they represent an initial collection of key features (necessary for the TaRDIS use cases to access the functionalities illustrated in the subsections 3.2-3.4 below) and they will evolve throughout the next iterations of this deliverable (D3.3, D3.5).

The description of a well-designed API encompasses a clear definition of its purpose and scope and an understanding of the specific functionalities it needs to expose and the types of interactions to be supported. While consolidating these initial sketches, TaRDIS plans to use standard API specification formats such as OAS (OpenAPI Specification),[22] RAML (RESTful API Modelling Language),[23] AsyncAPI,[24] or a combination of formats that suits a clearer explanation.

These API sketches below are divided between those conceived for internal services (Section 3.1.1), those conceived for perimeter services (Section 3.1.2), and those giving access to machine learning functionality (Section 3.1.3). The APIs deal with aspects such as the creation and validation of a swarm protocol, the creation of a role in a swarm protocol, creation of a workflow to handle that role, finding individuals running specific workflows or that are part of a specific role, creating communication overlays and channels, and handling events and messages.

[22] https://swagger.io/specification/
[23] https://raml.org/
[24] https://www.asyncapi.com/

Funded by the European Union

© 2023-2025 TaRDIS Consortium

3.1.1 Event-Driven API for Internal Services: Instantiating a TaRDIS Swarm

The following API sketches are intended to support the development of internal TaRDIS services (Section 2.1.1).

Defining a Swarm Protocol

Defines the protocol (set of rules) that will be the core of the swarm.

`defineSwarmProtocol(options)`

Parameters:

- `options: SwarmProtocolOptions`

Return:

- `SwarmProtocol`

Checking the Correctness of a Swarm Protocol

Checks the state of the swarm protocol.

`checkSwarmProtocol(protocol)`

Parameters:

- `protocol: SwarmProtocol`

Return:

- `SwarmProtocolState`

```
{
        state: String,
}
```

Creating a Workflow for a Role

Registers a workflow, which could be specific for a given role, or general (if the role is omitted).

`createWorkflow(protocol,role?)`

Parameters:

- `protocol: SwarmProtocol`
- `role: String`

Return:

- `Workflow`

```
{
        id: UUID,
        State1: Object,
        State2: Object,
        …
}
```


© 2023-2025 TaRDIS Consortium

Finding running instances of a workflow

Finds instance of a registered workflow by workflow instance ID

`findWorkflow(workflow, id)`

Parameters:

- `workflow: Workflow`
- `id: UUID`

Return:

- `Workflow | null`

Finds instances of registered workflows by desired workflow state.

`findWorkflows(workflow, cutoff?, states[]?)`

Parameters:

- `workflow: Workflow`
- `cutoff: Milliseconds`
- `states[]: WorkflowState[]`

Return:

- `Workflow[]`

3.1.2 Event-Based Input-Output API for Perimeter Services

The following API sketches are intended to support the development of perimeter TaRDIS services (Section 2.1.2).

Creating a Communication Overlay

Creates the communication overlay for a certain workflow. We can have different communication protocols for different workflows.

`createCommunicationOverlay(workflow, options)`

Parameters:

- `workflow: Workflow`
- `options: CommunicationOverlayOptions`

```
{
        routing: Object,
        membership: Object,
        sampling: Object,
        dissemination: Object,
        resourceLocation: Object,
        type: CommunicationOverlayType
        {
                Kadmelia,
                Chord,
```


© 2023-2025 TaRDIS Consortium

```
            HyParView,
            Scamp,
        }
    }
```

Return:

- `CommunicationOverlay`

Creating a Channel

Creates a channel on top of the communication overlay.

`createChannel(options, overlay)`

Parameters:

- `options: ChannelOptions`
```
{
    type: ChannelType
    {
        P2P,
        Multicast,
        Broadcast,
    }
}
```
- `overlay: CommunicationOverlay`

Return:

- `Channel`

Creating an Event

Register events on a channel.

`createEvent(channel, event)`

Parameters:

- `channel: Channel`
- `event: Event`

Emitting an Event

Emits the event on a channel.

`emitEvent(channel, event, data)`

Parameters:

- `channel: Channel`
- `event: Event`
- `data: Object`

© 2023-2025 TaRDIS Consortium

Subscribing to an Event

Subscribes to an event.

`subscribeToEvent(channel, event, handler)`

Parameters:

- `channel: Channel`
- `event: Event`
- `handler: (data) => {}`

Unsubscribing from an Event

Unsubscribes from events.

`unsubscribeFromEvent(channel, event)`

Parameters:

- `channel: Channel`
- `event: Event`

Closing a Channel

Closes a communication channel.

`closeChannel(channel)`

Parameters:

- `channel: Channel`
- `event: Event`

Sending a Message

Send a message through a channel.

`sendMessage(channel, message)`

Parameters:

- `channel: Channel`
- `message: Object`

Subscribing to Messages

Subscribe to messages on a channel.

`subscribeToMessages(channel, handler)`

Parameters:

- `channel: Channel`
- `handler: (message) => {}`

Funded by the European Union

© 2023-2025 TaRDIS Consortium

Unsubscribing from Messages

Unsubscribes from messages on a channel.

```
unsubscribeFromMessages(channel)
```
Parameters:
- `channel: Channel`

3.1.3 Machine Learning APIs

Loading a Model

Loads an ML model to the system.

```
loadModel(model)
```
Parameters:
- `model: Object`

3.2. Analysis and Verification Facilities

3.2.1. Specifying and Verifying Communication Behaviour - T4.1

3.2.1.1. Correctness-By-Construction Guarantees for Internal TaRDIS Applications

The workflow-based communication used by internal TaRDIS applications will guarantee deadlock-freedom by construction, provided that the individual workflows of all components are compatible with each other –- e.g. because they are projected from an overall correct swarm protocol (as outlined in the description of TaRDIS internal services, swarm protocols, and workflow, Section 2.1.1), or because they are analysed with one of the techniques for verifying communication behaviour (Section 3.2.1.2).

Since the TaRDIS swarm protocols allow cycles, correctness-by-construction cannot always guarantee process termination; however, the processes will terminate if:

1. swarm participants are available for each role and act when it is their turn, and
2. the protocol contains reachable final states and each swarm participant only selects available branches for a bounded number of times.

In addition, swarm protocols guarantee eventual consensus between all participants on the sequence of states visited, under the condition that the protocol definition fulfills a set of well-formedness conditions. These are checked using an API (outlined e.g. in the Actyx use case in section 4.1.1), which is typically used as part of the unit tests of the application.

```
const swarmProtocol = TaRDIS.defineSwarmProtocol(...)
const result = TaRDIS.checkSwarmProtocol(swarmProtocol)
expect(result).toEqual({ type: 'OK' })
```


© 2023-2025 TaRDIS Consortium

3.2.1.2. Communication Behavioural Properties

A primary goal of T4.1 is to develop innovative techniques based on behavioural types[25] [26] — specifically, typestates[27] and (multiparty) session types.[28] [29] These techniques will be used to assess if programs using the TaRDIS APIs satisfy desirable communication behavioural properties, chiefly focusing on internal TaRDIS services based on workflows (Section 2.1.1). The communication behavioural properties of interest include:

1. **Communication safety**: the exchanged data in a workflow-based communication (protocol) adheres to the expected types, ensuring the absence of any type errors.
2. **Deadlock-freedom**: a group of TaRDIS services involved in a workflow-based interaction will never get stuck.
3. **Termination**: TaRDIS services involved in a workflow-based interaction will terminate finitely.
4. **Never-termination**: a workflow-based interaction between TaRDIS services will continue indefinitely.
5. **Liveness**: in a workflow-based interaction between TaRDIS services, an event awaited by at least one service will be eventually emitted (i.e. all services will be able to progress).
6. **Protocol conformance and completion**: in a workflow-based interaction, valid sequences of calls of an API's methods can be defined with a typestate specification (henceforth called protocol). The code can be statically type-checked to ensure that services following the protocol never generate run-time errors like null-pointer exceptions (a safety property known as protocol conformance). Moreover, in the absence of divergent computation, it is also possible to ensure that the service code completes the intended workflow protocol (a weak liveness property known as protocol completion). In the context of a communication protocol, local conformance of all participants ensures the network's overall conformance to the initial protocol. Likewise, when considering a message-passing process, protocol conformance guarantees that the process behaves conforming to its declared types.

Based on behavioural type system methodologies, these behavioural properties will be verified utilising exhaustive static reasoning methods, such as static type checking and model checking, whenever feasible. We plan to combine three main approaches:

1. **Model checking swarm-protocol-level and workflow-level behavioural properties**, such as communication safety, deadlock-freedom, termination, never-termination, and liveness, as modal μ-calculus formulas, using tools such as mCRL2;[30]
2. **Type checking API usage against workflow specifications**, to check/enforce that client code interacting with the TaRDIS API correctly follows the workflow protocol;

---

[25] Hüttel et al: Foundations of Session Types and Behavioural Contracts. ACM Comput. Surv. 49(1): 3:1-3:36 (2016). https://doi.org/10.1145/2873052
[26] Ancona et al, "Behavioral Types in Programming Languages", Foundations and Trends® in Programming Languages: Vol. 3: No. 2-3, pp 95-230 (2016). http://dx.doi.org/10.1561/2500000031
[27] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," in IEEE Transactions on Software Engineering, vol. SE-12, no. 1, pp. 157-171, Jan. 1986. https://10.1109/TSE.1986.6312929
[28] Honda, Kohei, Vasco T Vasconcelos, and Makoto Kubo. 1998. 'Language Primitives and Type Discipline for Structured Communication-Based Programming'. In ESOP (1998). https://doi.org/10.1007/BFb0053567
[29] Honda, Kohei, Nobuko Yoshida, and Marco Carbone. 2016. 'Multiparty Asynchronous Session Types'. Journal of the ACM 63 (1): 1–67. https://doi.org/10.1145/2827695
[30] https://mcrl2.org/

Funded by the European Union

© 2023-2025 TaRDIS Consortium

3. Use workflow specifications to **monitor the use of the TaRDIS communication APIs**, only allowing, for each client, the sequences of requests prescribed by the workflow.

To this end, we work towards extending the core theories of session types and multiparty session types by incorporating features that allow for the application of behavioural types in supporting swarm protocols:

1. CLASS,[31] the first proposal for a foundational language able to flexibly express realistic concurrent programming idioms, features a type system ensuring all the following three key properties: CLASS programs never misuse or leak stateful resources or memory, they never deadlock, and they always terminate. CLASS expressiveness is illustrated with several examples involving memory-efficient linked data structures, sharing of resources with linear usage protocols, and sophisticated thread synchronisation
2. Teatrino,[32] a toolchain that utilises asynchronous multiparty session types (MPST) with crash-stop semantics to support failure handling protocols, providing correctness-by-construction guarantees to the users of the TaRDIS workflow-based APIs even in the presence of crashes and failures.
3. Dynamically Updatable Multiparty Session Protocols,[33] an extension to multiparty session types with the ability to add unbounded participants dynamically during a protocol execution. This allows the possibility of specifying and verifying swarm protocols.

3.2.1.3. Join Patterns API with "Fair Matching" Guarantees

As a contribution to WP4 / T4.1, DTU is developing an API for distributed systems based on *join patterns*, with correctness-by-construction guarantees ensuring that complex combinations of incoming messages/events are processed in a correct and "fair" way (as outlined in the next paragraphs). This API will be evaluated as part of the Actyx use case (Section 4.1), in a perimeter service for monitoring the occurrence of certain combinations of events (such as interrelated machine fault notifications) within certain time frames.

Join patterns are a programming construct originally introduced by the join calculus,[34] superficially reminiscent of the pattern matching constructs available in many programming languages: a programmer can write a join pattern to succinctly express that an application must listen for certain combinations (patterns) of incoming messages, whose payloads must satisfy certain conditions (guards); if a desired combination of messages becomes available, then the program must continue its execution accordingly.

For instance, the pseudo-code below shows a join pattern construct for an online trading system that inspects combinations of incoming `Sell` and `Buy` messages; the trading system looks for combinations of 2 or 3 messages where the amount of stocks being sold (`sA`, `sA1`, `sA2`) covers the amount being bought (`bA`). *(For brevity, we omit payload data besides the stock amounts.)*

---

[31] Pedro Rocha and Luís Caires: Propositions-as-types and shared state. Proc. ACM Program. Lang. 5(ICFP): 1-30 (2021). https://doi.org/10.1145/3473584

[32] Adam D. Barwell, Ping Hou, Nobuko Yoshida, Fangyi Zhou: Designing Asynchronous Multiparty Protocols with Crash-Stop Failures. ECOOP 2023. https://doi.org/10.4230/LIPIcs.ECOOP.2023.1

[33] David Castro-Perez and Nobuko Yoshida. Dynamically Updatable Multiparty Session Protocols: Generating Concurrent Go Code from Unbounded Protocols. ECOOP (2023). https://doi.org/10.4230/LIPIcs.ECOOP.2023.6

[34] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In ACM-SIGACT Symposium on Principles of Programming Languages, 1996. https://doi.org/10.1145/237721.237805

© 2023-2025 TaRDIS Consortium

```
receive {
    case Sell(..., sA) & Buy (..., bA)                 if bA ≤ sA   ⇒
      // Code to execute upon match
    case Sell(...,sA1) & Sell(...,sA2) & Buy (...,bA)  if bA ≤ sA1+sA2  ⇒
      // Code to execute upon match
}
```

Observe that the pseudo-code above only specifies which messages are of interest, and what conditions apply to their payloads — without detailing all possible combinations arising from the order of arrival of such messages (e.g. a `Buy` may arrive before a matching `Sell`, or *vice versa*). The manual handling of all such message combinations would be required by more traditional programming APIs based e.g. on actors or sockets, leading to complicated, error-prone, and potentially inefficient code.

DTU is working on a prototype implementation of a join patterns API similar to the pseudo-code above (implemented in the Scala 3 programming language). The implementation is based on a novel optimised message matching algorithm that is showing promising performance, and is being proven correct and "fair" — in the sense that (unlike previous work) all incoming messages will be eventually consumed if they can be matched. This ongoing work is outlined in a recent workshop paper.[35]

3.2.1.4. Verified Control Plane API for Software-Defined Networking

In some constrained swarm scenarios (involving e.g. smart devices in a local network, for instance within the same factory or IoT devices in a same domestic network) the configuration and maintenance of an overlay network of swarm components (Section 3.4.1.1) may take advantage of the possibility of configuring the underlying data link network. Such a dynamic reconfiguration is made possible by software-defined networking (SDN) — and more specifically, the Open Source SDN standard P4[36] specifies a *control plane* API[37] for programmatically querying and updating the current data link network configuration. Building upon these open SDN standards, DTU is developing a verified API, called P4R-Type,[38] to statically ensure that network control plane queries and updates are compatible with the underlying network configuration. We are investigating the integration of such an API in the TaRDIS toolbox.

3.2.2. Specifying and Analysing Data Consistency - T4.2

Analyzing and checking data consistency comprises verifying two main properties – state convergence (according to a specific convergence policy, over a spectrum of consistency requirements) and data invariant preservation. To support these analyses it is necessary to extract from the application code an abstract specification amenable for automated verification. The usual approach is to annotate code with the consistency policy required for each resource, application-specific invariants and, for some analyses, the operations' pre and post-conditions. We present next an illustration of a basic annotation language used to

---

[35] P. Haller, A. Hussein, H. Melgratti, A. Scalas, A. Sébert, E. Tuosto. A New Take on Join Patterns. Presented at NWPT 2023. https://conf.researchr.org/details/nwpt-2023/nwpt-2023-papers/16/A-New-Take-on-Join-Patterns
[36] https://p4.org/
[37] https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html
[38] Jens Kanstrup Larsen, Roberto Guanciale, Philipp Haller, and Alceste Scalas. 2023. P4R-Type: A Verified API for P4 Control Plane Programs. Proc. ACM Program. Lang. 7, OOPSLA2, Article 290 (October 2023), 29 pages. https://doi.org/10.1145/3622866

Funded by the European Union

© 2023-2025 TaRDIS Consortium

complement a Java interface for a (simple) tournament application with data invariants and operations' postconditions.[39]

```
@Inv("forall(Player:p, Tournament:t) :-
      enrolled(p, t) ==> player(p) and tournament(t)")
@Inv("forall(Tournament:t) :- active(t) => nrPlayers(t) >= 1")

@Inv("forall(Player:p,Tournament:t) :- nrPlayers(t) <= Capacity")

public interface TournamentApp {
 @True("player(p)")
 addPlayer(Player p);

 @False("player(p)")
 remPlayer(Player p);

 @True("tournament(t)")
 addTourn(Tournament t);

 @False("tournament(t)")
 remTourn(Tournament t);

 @True("enrolled(p, t)")
 @Increments("nrPlayers(t),1)")
 enroll(Player p, Tournament t);

 @False("enrolled(p, t)")
 @Decrements("nrPlayers(t),1)")
 disenroll(Player p, Tournament t);

 @True("active(t)")
 beginTourn(Tournament t);

 @False("active(t)")
 finishTourn(Tournament t);
}
```

In this approach an invariant is described by a first-order logic formula. More formally, it assumes the invariant is an universally quantified formula in prenex normal form (i.e., the formula has the shape $\forall x, \varphi(x)$ where φ is quantifier-free). The operations' postconditions are defined by basic clauses that abstract the operation effects.

Another possible alternative for the annotation language would be to use VeriFx,[40] which is a high-level functional programming language for defining and automatically verifying safety properties. A relevant feature of the language is the inclusion of a proof construct to express generic safety properties. For each proof, VeriFx automatically derives proof obligations and discharges them using SMT solvers (these solvers try to automatically determine whether or not a given formula is satisfiable). Therefore, VeriFx can be used, not only as an annotation language, but also as a push-button verification tool for data consistency properties and other safety properties.

---

[39] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, Marc Shapiro. Putting consistency back into eventual consistency. EuroSys 2015.
[40] Kevin De Porre, Carla Ferreira, Elisa Gonzalez Boix. VeriFx: Correct Replicated Data Types for the Masses. ECOOP 2023.

© 2023-2025 TaRDIS Consortium

To allow an application to support different consistency policies for different resources, and crucially, to identify calls on commutative operations and gather information on calls that in distributed runtime featuring replicas can be anticipated (w.r.t. calls requiring global replicas coordination), Giunti et al.[41] developed a language-based static analysis methodology. The intuition is: locally permissible operations can be immediately executed; strongly consistent ones require coordination among replicated sites. So, locally permissible ones can be anticipated safely with respect to strongly consistent ones. For instance, in a banking application, withdraw requires global coordination but deposit can always be executed. The code would look like this:

```
class Account {
  balance : int weak [ this.balance≥0] ...
  def withdraw(amount : int) : Unit [amount>0]
  { this . balance -= amount }
  def deposit(amount : int) : Unit [amount>0]
  { this.balance += amount }
...
}
```

The annotation burden is small – requires only to define consistency policies for resources that may not be strongly consistent - and assertions (boolean expressions) for state variables and operation parameters. The outcome is information to optimise the runtime, anticipating the execution of operations that do not need to wait for others to complete before, since they are commutative and manipulate resources that only require weak consistency.

Notice that any of the three approaches described can be easily integrated in the TaRDIS API formalism and in the adopted languages.

3.2.3. Specifying and Analysing Security Properties - T4.3

There are largely two general tasks for the security of information:

1. Ensuring the secure communication/transmission of data as part of events
2. Ensuring the secure storage of data

Here secure stands for any security aspects we might have, where we shall basically support the following:

1. **Confidentiality**: a security policy that describes who can read particular data
2. **Integrity/Authentication**: a policy who can write/author particular data

Here *confidentiality* is normally a basic form of protection by encryption in case of transmissions, so that unauthorized people cannot read the data itself. This does not include full non-interference guarantees, i.e., somebody observing the network traffic may be able to link transmitted messages of the same participant, may link messages to particular protocol steps, and may thus learn about the truth value of conditions in statements, which may in turn leak information about confidential data. To achieve higher levels of confidentiality, it is necessary to introduce "decoy" traffic as it is done for instance in onion routing protocols. We do not envision doing this for most TaRDIS use cases, but we investigate if such channels should be added as an option.

[41] Marco Giunti, Hervé Paulino, António Ravara:
Anticipation of Method Execution in Mixed Consistency Systems. SAC 2023.

© 2023-2025 TaRDIS Consortium

Integrity here also has several aspects. In its most basic form, for transmissions this is ensured by simply ensured by cryptography to ensure authentication (e.g. suitable encryption or MACs) which guarantees only that an attacker cannot create data under somebody else's identity or change/influence the data produced by somebody else. However, the attacker may be able to intercept and replay messages. Disruptions/availability problems cannot be solved by cryptographic protocols, and naturally all communication structures must have some fault tolerance, but this is not regarded here as a security issue. The replay indeed may cause security problems, as an attacker may with this influence decisions, e.g., replaying an old event, the attacker may cause an action by an honest participant that the attacker should not be able to cause. We thus distinguish between non-injective and injective agreement between participants: in the latter case, there is an injective mapping of acceptance of messages by a recipient to the issuing of messages by a sender. We leave this as an option since the replay protection may not be needed if the data sufficiently ensures freshness.

Another aspect of integrity/confidentiality in communication is the identity of the endpoints. For instance, TLS is the most common example of a secure channel between endpoints where only one side is authenticated: typically, only the server has a certificate and proves its identity with respect to it, but the client typically does not. Anyway, we have a secure connection between the two for a given connection; the channel can then be used to authenticate the client by (securely) sending their password with the server if they are a registered user. We currently envision that such authentication mechanisms should rather be part of the protocol and not visible at the event/transmission level.

We thus require the following specification items for all events that are for secure transmission:

- **The policy about the issuer of the event**: who is allowed to issue the event, i.e., what guarantees about the author of the event will the recipients have? Default is untrusted, i.e., no guarantee about the issuer.
- **The policy about the recipients**: who is allowed to receive this event, i.e., what guarantees about the confidentiality of the event does the sender have. Default is public, i.e., everyone can see it.
- **A flag whether injective agreement must be ensured by the communication channel**; the default is no, so there is no replay protection

The policy about origin and destination of an event can be specified as a set of entities which can be individual users, devices, roles, groups etc. Let C (for confidentiality) and I (integrity) be two such sets of users, we can regard (C,I) as a policy for a piece of data or an event: it must be ensured that only members of C can read and only members of I can write. The default for C (public) is all entities, the default for I (untrusted) is the emptyset. We have the usual security lattices by the subset/superset relation on the C and I components.

3.2.3.1. Formats

Not all information in an event/message has necessarily the same security level: one may transmit public data along with secret data and untrusted data along with trusted data. To allow that such public or untrusted data can flow after transmission also into positions that are public or untrusted, respectively, we allow that a message is like a record-type data type where each field may have different security levels, the entire format has then the supremum of the levels of the fields, and can only be transmitted on a suitable channel.

The implementation of a format can be any way to serialize the data, e.g., XML, ASN1, JSON, etc., where only two properties must be satisfied:

© 2023-2025 TaRDIS Consortium

1. For each format, a given string can be parsed in at most one way (unambiguous)
2. For different formats (that are used in the same communication system), a given string can be parsed as at most one format (disjoint)

These requirements allow for some relative soundness result of the form: if there is an attack, then there (also) exists an attack where the attacker only sent "well-typed" messages. This rules out so-called *type-flaw attacks* where the attacker abuses confusion of messages of different types like "agent" vs. "key".

Again, with respect to privacy, we do not protect in general the information which format is transmitted, but just its concrete contents. For instance, if we have formats f1 and f2 representing two kinds of events in the system, one needs to generally assume that the attacker knows for a given message whether it contains an f1 or f2 format.

3.2.3.2. Security Analysis

The security analysis is now on the level of the application issuing and consuming events, and on the level of protocols that implement the communication of events.

On the application level, we shall check with information flow methods, that there is not explicit or implicit flow from data with a higher level to an event or memory location of a lower level (so confidential information cannot leak into public, and untrusted information cannot leak into trusted).

For the security protocols, we need to verify that they indeed provide the security guarantees that they claim to implement, e.g., that upon transmission they ensure the integrity and confidentiality from senders to receivers.

For each format in use, we need to check that their implementation satisfies the above properties (unambiguity and disjointness).

Moreover, are currently working on extending compositionality results that show: given the transmission protocol is proved secure for abstract data, and given the application satisfies information flow given a secure implementation of the protocol, then the entire system of application over channel is secure. The application of such a result to a concrete application requires only some static checks that the employed formats of application and channel are also sufficiently disjoint.

3.2.3.3. Specification of the setup

Note that it needs to be specified which protocols the application shall be using for the transmission of events. It may even be that different protocols are used for different kinds of events. This also ensures how cryptographic keys are set up in these cases for each participant that needs to encrypt/decrypt/verify anything as part of the channel implementation.

3.2.3.4. Specification of administration protocols

During operation, the groups of an application may change, e.g., we may have new users of an electric grid, or a user may change its group membership. In these cases, updates of the cryptographic keys may be necessary and the TaRDIS API should allow "events" that trigger respective key update protocols.

Funded by
the European Union

© 2023-2025 TaRDIS Consortium

3.2.4. Deployment and Orchestration Integration - T4.4

In Task T4.4, the correctness of the federated learning algorithm implemented by PTB-FLA (whose API is described below, as part of T5.1, Section 3.3.1.1) has been verified for deadlock-freedom and termination.[42] This provides correctness-by-construction guarantees to the users of the API, ensuring that the federated learning algorithm will never get stuck. Future work of T4.4 includes proving that the federated learning algorithm will always converge to a result.

3.3. Artificial Intelligence and Machine Learning APIs

3.3.1. AI/ML Programming Primitives - T5.1

3.3.1.1. Python Test Bed for Federated Learning Algorithms (PTB-FLA) API

T5.1 plans to integrate in TaRDIS the PTB-FLA API, based on the federated learning research described in several recent papers.[43] [44] [45]

The PtbFla API is offered by the class PtbFla, which comprises the constructor, the two generic FLAs, and the destructor:

```
1.  PtbFla(noNodes, nodeId, flSrvId=0)
2.  ret fl_centralized(sfun, cfun, ldata, pdata, noIters=1)
3.  ret fl_decentralized(sfun, cfun, ldata, pdata, noIters=1)
4.  PtbFla()
```

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
- 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, `ldata` is a machine learning model, whereas `pdata` is a training data that is used to train the model. Normally, the

---

[42] 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, https://doi.org/10.1109/ZINC58345.2023.10173859

[43] 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, https://doi.org/10.1109/ZINC58345.2023.10173859

[44] M. Popovic, M. Popovic, I. Kastelan, M. Djukic, I. Basicevic, "A Federated Learning Algorithms Development Paradigm," ECBS 2023 (to appear). arXiv:2310.05102 [cs.DC] https://doi.org/10.48550/arXiv.2310.05102

[45] I. Prokić, S. Ghilezan, S. Kašterović, M. Popovic, M. Popovic, I. Kaštelan, "Correct orchestration of Federated Learning generic algorithms: formalisation and verification in CSP," ECBS 2023 (to appear). arXiv:2306.14529 [cs.DC] https://arxiv.org/abs/2306.14529

© 2023-2025 TaRDIS Consortium

testbed instances only exchange ldata and they never send out `pdata` (that is how they guarantee the training data privacy). The `pdata` is only passed to callback functions within the same process instance to immediately set them in their working context.

Cross-references: Provides service to (i.e., is used by). PTB-FLA API might be used by LSTM and DRFL models. Roadmap: In October 2023, NKUA and UNS started collaboration on testing the testbed. The idea is to first test the testbed with a simple LSTM model in a few nodes both in its centralized and decentralized versions. Then, at later stages of the TaRDIS project, it might be useful to also host in the testbed the training of DRFL models for the EDP use case.

Cross-reference: Uses service of (i.e., depends on). At present, PTB-FLA uses its own message passing solution that is implemented in the module mpapi.py, which in turn uses the Python module `multiprocessing`.[46] Therefore, PTB-FLA API may be classified as a perimeter TaRDIS service. However, since mpapi.py API comprises functions (sendMsg, rcvMsg, broadcastMsg, and rcvMsgs) that seem to be close to API offered by WP6, it might be feasible to adapt mpapi.py to use the latter API, in this case PTB-FLA might become an internal TaRDIS service (assuming that WP6 API would be a service of that type).

Formalization and Verification. PTB-FLA generic algorithms have been formalized using CSP (Communicating Sequential Process) calculus and verified using model checker PAT (Process Analysis Toolkit).[47] Two properties were checked: deadlock freedom (safety) and termination (liveness). It seems appropriate to mention that TaRDIS formal verification tools should use a compositional approach, and if they do, then they should take these two properties for granted for PTB-FLA, rather than keep rechecking them at application build time. It is also worth noting that converting Python to CSP was done manually, but could be automated, and since PAT is maintained by the University of Singapore, it might be possible to arrange its usage from the TaRDIS toolkit.

3.3.1.2. Flower-based Federated Learning (FFL) APIs

Here we describe the candidate APIs that relate to the work on Flower-based FL (FFL) implementations in the context of Task 5.1 carried out by FTN, which is complementary to PTB-FLA (FFL is cloud-based whereas PTB-FLA is targeting smart IoTs in edge systems). Each of the three FFL APIs described below might be of interest and "called", e.g., from the Actyx use case - manager view application (Section 4.1.3).

3.3.1.2.1. FFL training models API

This API focuses on offering a list of algorithms, implemented in the Flower framework, that can be called for the purposes of TaRDIS use cases. Currently, it contains the implementation for FedAvg and Personalized Federated Learning with Moreau Envelopes. The API will also offer a solution for anomaly detection with noisy labels, that is of interest for the Actyx use case, App 1. Distributionally robust FL algorithms are also considered. The complete list of the algorithms will evolve according to use cases' needs.

The Flower framework offers tools for defining a ML model and the basic skeleton for building FL strategies. The algorithms in the API will use the structure provided by Flower, and implement custom strategies for ML algorithms.

Different client selection protocols may be implemented as well. Moreover, some of these techniques could possibly be integrated with the FLaaS system by Telefonica (Section 4.4).

[46] https://docs.python.org/3/library/multiprocessing.html
[47] https://pat.comp.nus.edu.sg/

© 2023-2025 TaRDIS Consortium

The provided FL algorithms will naturally support pre-trained models but will also be constructed so that incremental model retraining is possible, which is an important feature to enable incremental model enhancement, as it is the case, e.g., with the Actyx use case (Section 4.1). The FL ML pseudocode can be viewed as follows:

```
Begin:
      Load data and specify input parameters
      Perform preprocessing
Set up the model
      Begin loop:
            Re-train the model with FL
      Perform inference
```

3.3.1.2.2. FFL Input data preprocessing API

In order to bridge the gap between the raw input data and FL ML algorithms, an API for data preprocessing and preparing is also planned. It will enable the transformation of the data into the format suitable for analysis by the ML model. Besides applying usual data preparation techniques for profiling, cleansing, and transformation, it may also support additional features, such as pseudo-labeling, required, e.g., by the Actyx use case (Section 4.1).

3.3.1.2.3. FFL ML inference and evaluation API

Inference represents an important component in FL, as it enables gaining output for the relevant data on a trained model. Besides inference, the FL ML library will provide evaluation, in the form of the corresponding metrics of the gathered results where possible.

3.3.2. AI-Driven Planning, Deployment, and Orchestration - T5.2

3.3.2.1. Reinforcement learning orchestration API

This API provides the functionality to create, manage, and connect to the TaRDIS framework the reinforcement learning agents for computer network orchestration. The action space for the agents is composed of task offloading decisions.

The initial training of the agent is done by interaction with a simulation environment that should exhibit properties as close as possible to the real network to orchestrate. Then, the agents should be deployed in a safe and controlled set of real hardware. Lastly, the agents are deployed for continual learning and orchestration of the real computer network.

The reward function in the context of reinforcement learning for computer network orchestration is a crucial component that quantifies the success of a given action or sequence of actions taken by the agent. It is a function that maps each state-action pair to a real number, which represents the desirability of that action in that state. In the context of network orchestration, the reward could be based on various factors such as latency, throughput, cost, or any other relevant metric. The aim of the agent is to learn a policy that maximizes the cumulative reward over time.

3.3.2.1. Reward Function Configuration

To configure the reward function, one needs to define the specific metrics that the function will consider. For instance, the reward function could be set to prioritize latency, in which case it would assign higher rewards to actions that minimize latency. Conversely, if the priority is to reduce costs, the reward function could be set up to allocate higher rewards to actions that result in cost reductions.

© 2023-2025 TaRDIS Consortium

One can also configure a reward function that balances multiple metrics. For example, a reward function could be set to assign high rewards to actions that both minimize latency and cost. This would require careful tuning of the reward function to ensure that the trade-offs between the different metrics are appropriately balanced.

In general, the configuration of the reward function should be guided by the specific objectives of the network orchestration. The reward function should be set up in such a way that it encourages the reinforcement learning agent to take actions that align with these objectives.

3.3.2.1.1. Reward Function Configuration

```
class RewardFunction:
    def __init__(self, weights):
        self.weights = weights

    def calculate_reward(self, state, action):
        # calculate metrics based on state and action
        latency = calculate_latency(state, action)
        cost = calculate_cost(state, action)
        # ... calculate other metrics

        # calculate reward based on metrics and weights
        reward = self.weights['latency'] * latency + self.weights['cost'] * cost
        # ... add other metrics to the reward calculation

        return reward
```

3.3.2.1.2. Agent Call Function

```
def agent(s_t, r):
    # s_t represents the current state
    # r represents the reward function
    a_t = policy(s_t, r) # a_t represents the action taken by the agent
    return a_t
```

The configuration of the simulation environment is detailed next. The agents can be centralized and decentralized. The following sections detail the api with Tardis for the programmer.

3.3.2.2. Deployment interface

As shown in the pseudo-code above, the agent inference and the computation of the reward both depend on the state of the simulation. In the remainder of this section, we will detail the state space for a centralized version of the agent and a decentralized version of the agent.

3.3.2.2.1. Centralized version

The state space of the centralized version of the simulation consists of the tuple:

```
<allNodeIds, allQueueSizes, allProcessingPower, allNodeLayers, allPositions,
allMaxBandwidths, allTransmissionPowers, allAverageCompletionTimes>
```


© 2023-2025 TaRDIS Consortium

`allNodeIds` - An array with the identifiers of all the nodes in the network; all the other arrays respect the order of the IDs in this array. This means that the entry 'n' in any other field represented by an array in the global state would pertain to the node with the ID stored in entry 'n' of this array.

- `allQueueSizes` - An array storing the last known queue size of all the nodes in the network.
- `allProcessingPower` - An array storing the latest known processing power of all the nodes in the network.
- `allNodeLayers` - An array storing the layer every node in the network belongs to
- `allPositions` - An array storing all the known positions of all the nodes in the network.
- `allMaxBandwidths` - An array storing the bandwidth of the channels of the centralized node to all its neighbors.
- `transmissionPower` - The transmission power of the central node's antenna

3.3.2.2.2. Decentralized version

The state space of the decentralized version of the agent consists of the tuple:

```
<nodeId, maxQueueSize, neighbourQueues, maxProcessingPower, position,
bandwidthCapacity, transmissionPower>
```

- `nodeID` - The identifier of the node in the current simulation
- `maxQueueSize` - The capacity of the node's task queue
- `neighbourQueues` - An array storing the latest knowledge the node has of the neighbors' queue sizes.
- `maxProcessingPower` -The maximum processing power of the node
- `position` - The coordinates of the node in a 100x100 grid
- `maxBandwidth` - The bandwidth capacity of the channels the node has access to.
- `transmissionPower` - The transmission power of the node's antenna.

3.3.2.3. Initial training simulation

To develop the simulation interface we utilized the Peersim simulation engine,[48] which allows for large-scale simulations with high dynamicity to be run. Furthermore, the Peersim Engine allows for customizable and in-depth configuration of the simulation through configuration files. A feature we find very useful.

In the subsequent section, we will delve into a comprehensive explanation of the simulation configurations.

3.3.2.3.1. Configuration of the simulation

Before beginning to explain the possible configurations of the different parts of the network we need to go over what these parts are. The simulation works by first generating a set of nodes that can be of two different types, Workers and Clients. The Clients act as task generators of work and collect metrics on the conclusion of tasks. Whereas the Worker Node acts as a processing node for said tasks, meaning the Worker node is the one doing the

[48] Alberto Montresor and Mark Jelasity. PeerSim: A scalable P2P simulator. In Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09), pages 99–100. Seattle, WA, September 2009.

© 2023-2025 TaRDIS Consortium

actual computing of the tasks. The Worker nodes also may host a controller component, which is a protocol that will run on the same node of the workers and is responsible for collecting information about the node's neighborhood and sending such information in one-hop broadcasts.

The challenge of task offloading encompasses various configurations, and we have implemented some of these specific instances. These include Online-Binary task offloading, where the decision to offload the next available task is made dynamically; and Batch Task offloading, where, at each time step, decisions are made for every task that arrived in the last time-step regarding offloading. Furthermore, we are working on implementing a dependency-cognizant task offloading simulation. The selection of these modes is done by defining a base file that outlines the appropriate protocols for each instance of the problem. Beyond determining the simulation mode, the configurability extends to a range of global parameters applicable to all instances, in the remainder of this section we will be going over the different configurations that are allowed.

The global configurations encompass six domains: Global Configurations, Worker-specific configurations, Controller-specific configurations, Client-specific configurations, Cloud-specific configurations, and lastly, topology and network-specific configurations.

3.3.2.3.2. Global Configurations

Overall configurations of the simulation

Size of the Network. Defines the number of nodes in the simulation, due to the properties of Peersim, if we have stipulated size N to the simulation then there are N workers and N Clients.

```
SIZE 10
network.size SIZE
```

Number of times the simulation is executed. The constant 'CYCLES' defines the total number of complete cycles (ticks in the simulation) that a simulation starting from 0 takes to end. We should note the actual variable that sets this value is 'simulation.endtime', but for convenience

The constant 'CYCLE' is used to define the number of ticks to reschedule the making of an offloading decision. For example, if the value is left at one an offloading decision will be made every time step.

```
CYCLES 1000
CYCLE 1

# You only need to set CYCLE and CYCLES, not recommended to alter the
# settings below directly. They are shown for informative purposes only.
...
simulation.endtime CYCLE*CYCLES
```

Bounds of the delay a message can have. Setting MINDELAY and MAXDELAY will allow messages to randomly be delayed when being delivered. The delay will be such that MINDELAY <= delay <= MAXDELAY and MAXDELAY == MINDELAY == 0 means no delay.

```
MINDELAY 0
MAXDELAY 0
```


© 2023-2025 TaRDIS Consortium

The probability of a package/message being lost. Allows messages to be lost.

```
DROP 0
```

Define the indexes of the nodes that have the controller function. This parameter specifies the indexes of the nodes on the global network vector that have a controller function.

```
CONTROLLERS 0,1,2
```

The flag that manages the existence of the Cloud. This flag can take values of either 1 or 0. If the flag is set to 1 an extra node not included in the nodes specified from 'SIZE' is created. This means that if 'SIZE 10' and 'CLOUD_EXISTS 1' then there is a total of 11 nodes.

```
CLOUD_EXISTS 1
```

Configurations of the Simulation Flow and the PettingZoo environment

Utility Reward. a parameter of the reward function acts as a weight in the expression that computes the utility of a reward. This parameter receives an integer. It is directly used by the environment and not the simulation.

```
protocol.mng.r_u 1
```

Delay Weight. A parameter of the reward function acts as a weight in the expression that computes the cost associated with the delay induced by taking an action. This parameter receives an int. It is directly used by the environment and not the simulation.

```
protocol.mng.X_d 1
```

Overload Weight. A parameter of the reward function acts as a weight in the expression that computes the cost associated with node overloading (overloading of a node happens when the node has too many tasks assigned and starts losing tasks) induced by taking an action. This parameter receives an int. It is directly used by the environment and not the simulation.

```
protocol.mng.X_o 150
```

Configurations of the Client

All the work generated on the simulation is in the form of applications, which in turn consist of groups of tasks. The Clients generate applications, and the types of applications generated have different properties depending on the type of simulation being implemented. In the case of Binary-Online and batch offloading an application consists of a single task. In a simulation with dependencies, an application consists of a Directed Acyclic Graph (DAG) of tasks.

A task consists of an amount of instructions to be processed, a total data size inputted, and a cost in CPU cycles per instruction.

Similarly to the tasks, there can also be multiple types of DAGs, in the simulations with dependencies the DAG type is selected randomly whenever the client is generating an application. A DAG is modeled as a set of tasks, in which the type is randomly selected on creation, a list of dependencies that must always start with task 0 and end in the last task.

© 2023-2025 TaRDIS Consortium

For each of the DAG or task types, all the configurations must be specified, otherwise an error is thrown on environment creation. In the case of the dependency-less simulations, there must be only one DAG type with one vertex and no edges.

Client's parameters

Task Arrival Rate per Client. This parameter acts as the event rate of an exponential distribution, which rules when the next application to be sent to a given node will be generated. Each client keeps track of when to send a new application to each of the workers it can see independently. The computation of the time for the next event is done by inverting the cumulative distribution function of the exponential distribution and sampling a uniform distribution between 0 and 1, by applying the inverse distribution on the sampled value we obtain the time for the next event corresponding to the sampled value.

```
protocol.clt.taskArrivalRate 0.1
```

Task Parameters

Max Possible deadline. This parameter allows for defining the minimum deadline. To disable deadlines, set this parameter to be less or equal to zero.

```
protocol.clt.minDeadline 100
```

Number of Tasks. This parameter specifies the total number of task types in the simulation.

```
protocol.clt.numberOfTasks 2
```

Ratios of each task type. The ratio at which each task type is selected is specified through this parameter. The actual values used do not matter as the weights will be scaled and converted into probabilities.

```
protocol.clt.weight 4,6
```

Average Number of Cycles in an Instruction. The number of cycles per instruction of each task type. This parameter is used in two ways:

- In computing the reward function. Specifically, affects the delay function and represents the execution cost of the tasks.
- It is considered in computing the time it takes for a simulation to finish a task.

```
protocol.clt.CPI 1,1
```

Byte Size of Task. The byte sizes of each task type. This parameter is measured in Mbytes and used in computing the cost in time of communication when offloading tasks and impacts the communication cost in the Reward function.

```
protocol.clt.T 150,100
```

Number of Instructions per Task. The byte sizes of each task type. This parameter, similarly to the CPI, is used in two ways:

- In computing the reward function. Specifically, it affects the delay function and represents the execution cost of the tasks.
- It is considered in computing the time it takes for a simulation to finish a task.

```
protocol.clt.I 200e6,250e6
```


© 2023-2025 TaRDIS Consortium

DAG Parameters

Number of DAGs. This parameter specifies the total number of DAG types in the simulation.

```
protocol.clt.numberOfDAG 2
```

Ratios of each DAG type. The ratio of each DAG type. The actual values used do not matter as the weights will be scaled and converted into probabilities.

```
protocol.clt.dagWeights 4,6
```

Edges. This parameter specifies the edge configurations of each of the DAG types. an edge is represented by a string of the form predecessorVertice->successorVertice; different edges within a DAG are separated by a ','. The edge sets of different DAGs are separated by a ';'. Furthermore, the edges of a DAG must obey the following rules:

- Task 0, the initial task, must be a predecessor to all tasks and have no predecessors of its own.
- If there are n+1 vertices the vertice of index n, must be a successor to all tasks and have no successors of its own.

```
protocol.clt.edges 0->1,1->2,2->3,3->4,4->5,5->6,6->7,0->8,8->7,7->9
```

The number of vertices in the DAG. This parameter indicates the total amount of vertices in the DAG. This value must be the value of the highest indexed vertice in the edges plus one.

```
protocol.clt.vertices 10
```

Configurations of the Worker

The workers are defined in sets of nodes with the same properties, which we call layers. The layers are organized in a vector and for each layer there is a processing power that consists of the number of available cores multiplied by the frequency of the worker's CPU, and a maximum queue size. We can add heterogeneity to the nodes in a layer by specifying a deviation term to the frequency of each CPU in that layer.

Nodes can only communicate with nodes on its layer, the layer immediately below and the one immediately above, for example, a node in layer index 1, would only be able to communicate with nodes in the layer index 0 and index 2. This can be overridden by manually specifying the links of the network, we shall explain how to do this later. Only nodes in layer 0 can communicate directly with the clients.

Specify the number of layers in the simulation, This flag informs the simulation of the total number of layers to be created. The value of this flag must be equal to the number of entries in the 'NO_NODE_PER_LAYER'.

```
NO_LAYERS 2
```

Number of nodes in each layer, this flag defines the number of nodes in each of the layers of the simulation. The sum of the nodes in all the layers must total the value in SIZE. Each entry is separated by a ',' and indicates the number of nodes to be put on the layer of index equal to its position, for example, if we have the configuration '5,0' then we would have 5 nodes in the layer of index 0 and 1 node in the layer of index 1.

© 2023-2025 TaRDIS Consortium

```
NO_NODES_PER_LAYERS 5,1
```

Number of Cores in Worker CPU. This parameter specifies the number of cores a node in each layer will have, the value for each layer is separated by a ','. For example, given the value '4,8', the nodes in layer index 0 will have 4 cores and the nodes in layer index 1 will have 8 cores. The parameter is used for two things:

- In computing the reward function. Specifically, affects the delay function and represents the execution cost of the tasks.
- It is considered in computing the time it takes for a simulation to finish a task.

```
NO_CORES 4,8
```

Frequency of Worker CPU. This parameter is measured in instructions/second, and specifies the base frequency a node in each layer will have, the value for each layer is separated by a ','. For example, given the value '1e7,3e7', the nodes in the layer index 0 will have a frequency of 1e7 instr/second and the nodes in the layer index 1 will have a frequency of 3e7 instr/second. The parameter is used for two things:

- In computing the reward function. Specifically, affects the delay function and represents the execution cost of the tasks.
- It is considered in computing the time it takes for a simulation to finish a task.

```
FREQS 1e7,3e7
```

Variations between frequencies of nodes in the same layer This parameter specifies the variation of the frequency between nodes in each layer. This means that a node that is created in a layer with frequency 1e7 and variation 1e3 can be created with a frequency between [1e7-1e3, 1e7+1e3]

```
VARIATIONS 1e3,1e3
```

Maximum Queue size. This parameter is used multiple times when computing the reward function and is used as the threshold for a node to overload and start dropping tasks. Similarly to the frequency and number of cores, it is specified for each layer as a list of maximum queue lengths separated by ','.

```
Q_MAX 10,50
```

Configuring the Topology

There are three ways of configuring the topology of the network. The first is the manual way, where we can specify a concrete topology by indicating the position of the nodes and their links, alternatively. The second is the automatic way, where we randomly place the nodes and they will be able to communicate with everyone in a neighborhood of user-defined radius. Lastly, it is possible to specify the position of the nodes and have them linked to every node in a neighborhood of a user-specified radius.

Randomize Positions Flag, this flag is the one that specifies whether the nodes are to be placed randomly or a topology was specified manually.

```
RANDOMIZEPOSITIONS true
```


© 2023-2025 TaRDIS Consortium

Specify the Positions of the nodes, the coordinates of the nodes are specified in one String with the format "X0,Y0;X1,Y1;...". The coordinates of each node are separated by a ';', so the coordinates of the first node are (X0, Y0) and the coordinates of the second are (X1, Y1).

```
init.Net0.POSITIONS
18.55895350495783,17.02475796027715;47.56499372388999,57.28732691557995;5.366
872150976409,43.28729893321355
```

Randomize Positions Flag, this flag is the one that specifies whether nodes are to be linked using the radius method or using the manual definition of the links.

```
RANDOMIZETOPOLOGY true
```

Specify the links between the nodes. This parameter allows for manually specifying the links between nodes. The parameter is of the form, 'node_idx,neigh0,neigh1,...' where the first entry is the node's index, then we list the indexes of the nodes it has links to. Entries are separated by a ';'. If a node has no neighbours, we must specify the node's index without any following indexes, for example '0;'.

```
init.Net1.TOPOLOGY 0,1;1,0,2;2,1;
```

The Radius of the neighbourhood. This parameter defines the radius of the neighbourhood of a node, the area in which a node knows all other nodes.

```
init.Net1.r 50
```

Configuration of the links between the nodes

The Bandwidth of a Link. This parameter is measured in Mhz and is used in computing the cost in time of communication when offloading tasks and impacts the communication cost in the Reward function. Currently, the bandwidth is equal for all links.

```
protocol.props.B 2
```

The Path Loss Constant of a link. This parameter is used in computing the cost in time of communication when offloading tasks and impacts the communication cost in the Reward function. Currently, the path loss constant is equal for all links.

```
protocol.props.Beta1 0.001
```

The Path Loss Exponent of a link. This parameter is used in computing the cost in time of communication when offloading tasks and impacts the communication cost in the Reward function. Currently, the path loss exponent is equal for all links.

```
protocol.props.Beta2 4
```

The Transmission Power of a node. This parameter is measured in dbm and is used in computing the cost in time of communication when offloading tasks and impacts the communication cost in the Reward function. Currently, all nodes have the same transmission power.

```
protocol.props.P_ti 20
```

Funded by the European Union

© 2023-2025 TaRDIS Consortium

Specify which flags have a link to the cloud. This parameter indicates for each layer if the nodes in that layer can access the cloud or not. If the value at the index of the layer is 1, then that layer can communicate with the cloud. Otherwise, if it is 0, then the nodes in the layer are barred from communicating directly with the cloud. For example, for the configuration '0,1', the nodes in the layer of index 1 would be able to communicate with the cloud, but the nodes in the layer of index 0 would not.

```
CLOUD_ACCESS 0,1
```

Configuration of the Cloud

We consider the cloud as a collection of virtual machines that allow for processing multiple tasks concurrently, one per VM. Each of these virtual machines is similar to a Worker in the sense that every time-step they will be able to process a given number of instructions. Each one will need to have processing power specified.

Number of VMs available to the Cloud - This specifies the number of VMs that can process concurrent tasks in the Cloud at the same time.

```
protocol.cld.no_vms 3
```

Processing Power of the VMs - This is the number of instructions that a VM in the cloud can produce in a time step.

```
protocol.cld.VMProcessingPower 1e8
```

These parameters are evolving and the most recent version will be available in the documentation of the github repository for the RL orchestration of TaRDIS.

3.3.3. Lightweight and Energy-Efficient ML Techniques - T5.3

At the moment, we foresee three techniques for making a ML model more lightweight and energy-efficient. All three techniques are related to the use of Deep Neural Networks (DNNs) - lightweight methods for “simple” ML models (e.g., Support Vector Machines, Random Forest, etc.) are out of scope of the project, since the DNNs require the most computational resources. Moreover, DNNs are currently state of the art and are going to be implemented most probably for all 4 use cases. In specific, the three methods are:

- **Early exit of inference**: This method can be implemented in a DNN that includes multiple hidden layers. Assuming that the ML model is already trained, during its inference phase the test/unseen data move through the input layer and are fed-forward to the DNN. The DNN has multiple hidden layers that may also span multiple architectural/hierarchical layers, i.e, some layers are in an IoT device, the sequential hidden layers are in an edge server, the final hidden and output layers are in the cloud. Thus, in order to make the inference of the ML model, data has to go through IoT to edge to cloud and through all the layers (computationally heavy and time-consuming). In the early exit of inference, several exit points (output layers) are included in the architecture of the DNN during its training. The DNN is therefore trained with the multiple output layers. During the inference, the output result provided in the first model exit (e.g., after 2 hidden layers in a mobile device) is accompanied with a confidence/accuracy score. In case that the latter score is acceptable, the model early exits and the data does not run through all the rest of the hidden layers, making it more energy and computationally efficient and faster. This method can be

© 2023-2025 TaRDIS Consortium

applied to both classification tasks (e.g., object detection) and regression tasks (e.g., orbit determination); preferably it is implemented in classification tasks. It requires:
  - The base ML model (model that the ML developer wants to transform to more lightweight)
  - Type (classification or regression)
  - Number of exits (number of early exits/outputs that will be included in the designed model)
  - Accuracy threshold (in order to locate the exits in the DNN architecture and train the model)
  - Finally, the new early-exit version of the DNN must have access to the training dataset

- **Knowledge Distillation (KD)**: this method targets to transform a large DNN model (with multiple hidden layers) to a smaller one that is more compact, more energy and computationally-efficient, without losing significant accuracy in the DNN output/prediction/*estimation. In this context we have a teacher model (the original DNN) and a student* model (the lightweight DNN). The training dataset is used to infer both the teacher and the student DNNs and then, the student uses as a label the output of the teacher model. This method can be applied to both classification and regression tasks (better results with classification tasks). It requires:
  - The base ML model (model that the ML developer wants to transform to more lightweight)
  - Type (classification or regression task); if regression, we possibly need as input from the user/ML developer a selection among several loss functions
  - Number of hidden layers (required by the developer for the student model)
  - Accuracy threshold (try to train the lightweight student network without dropping below an accuracy threshold)
  - The student and the teacher DNNs must have access to the training dataset

- **Pruning**: this technique practically sets to zero some of the weights in several neurons of a DNN that play a negligible role in the performance of the ML model. The main idea is that some of the NN connections will not be triggered, thus making the DNN more computationally- and energy-efficient, as well as speeding up the inference process. This method works well both for classification and regression tasks. It requires:
  - The base ML model (model that the ML developer wants to transform to more lightweight)
  - Compression rate (%) and Accuracy (%) compared to the original DNN. In general compression rate and accuracy should experience a trade-off
  - Speedup factor (e.g., make the inference 2 times faster)
  - Method that will be utilized (global magnitude, global gradient magnitude, laywise magnitude, laywise gradient magnitude)
  - The new DNN must have access to the training dataset

3.4. Data Management and Distribution Primitives

3.4.1. Decentralised Membership and Communication APIs - T6.1

The TaRDIS approach emphasises decentralised distributed architectures, potentially composed by a large number of independent computational components that interact among themselves - in a dynamic fashion - to ensure the operation of an application as a whole.

© 2023-2025 TaRDIS Consortium

To support this type of architecture, TaRDIS relies on several types of distributed (and decentralised) protocols that provide different abstractions (potentially with different guarantees) for TaRDIS applications. The distributed abstractions that we consider in TaRDIS are the following:

a) Overlay Networks (Section 3.4.1.1), which fundamentally define and maintain a logical network interconnecting the different components of a TaRDIS application, and that self-manages in face of changes on the system affiliation (components joining, leaving, or failing) and potentially to other dynamic aspects of the environment (e.g., reliability of a network link) or system (e.g., variations in the workload).
b) Communication Primitives (Section 3.4.1.2), which operate on top of overlay networks, provide the fundamental mechanisms that allow different components of a TaRDIS application to interact, exchange information, and coordinate. In the context of TaRDIS we consider several types of communication primitives, including point-to-point and several point-to-multipoint abstractions (e.g., broadcast, multicast, publish-subscribe). As detailed further ahead, these primitives can provide different types of guarantees for applications.
c) Distributed Storage Abstractions, which provide the fundamental mechanisms for managing the state of TaRDIS applications, including API for defining a distributed data model, and operations that both expose the state of the application or modify it. These abstractions, as detailed further ahead, can provide a wide range of guarantees and properties for applications.

While the specification of these distributed abstractions and their materialisation through a distributed protocol is provided by TaRDIS and can be freely implemented in any programming language or development ecosystem, concrete implementations of these abstractions will also be provided as part of the TaRDIS toolbox. In the following we discuss the properties and specific APIs that are currently planned to be provided and implemented by the TaRDIS team for each class of abstractions.

3.4.1.1. Overlay Network Specifications and APIs

Overlay network reside at the lowest level of our distributed protocols stack (we assume these operate above the operating system layer which provides a TCP/IP and UDP/IP stack with a standard Posix interface), at the most fundamental level, overlay networks provide a form of system membership management which allows components of a TaRDIS application to track (even if partially) other components of that application, enabling the construction of other primitives (communication and storage) on top of them.

The literature in peer-to-peer systems is rich in the proposal of several different types of overlay designs, which can be classified in terms of the type of topology they strive to build and maintain, either unstructured overlays (when the topology is random and impossible to predict a-priori) or structured overlays (when the protocol strives to enforce some known apriori topology invariants, typically based on probabilistic unique identifiers of nodes/processes in the overlay network). These topology properties allow some overlays to provide additional functionality for applications in addition to a form of membership tracking, consider for instance Distributed Hash Tables (DHTs), a specific form of a structured overlay network, which in addition to be able to provide to each node in the system a sample of other nodes currently in operation, also can provide application-level routing within the identifier space of the nodes that compose the overlay. Naturally, some functionalities that can be provided by overlay depend on their specific topology, however, in our API to manage overlay networks, we allow programmers to specify only the functionalities that the application requires from the overlay, and having the runtime select the better overlay (or overlays) to provide the target functionalities. Optionally, the programmer can indicate the specific overlay they wish to use (in addition to functionality) which allows the runtime to validate the

© 2023-2025 TaRDIS Consortium

selection at compile time. (A more extensive discussion on overlay designs and functionalities can be found in the TaRDIS deliverable D6.1)

In general, the API for managing overlay networks that TaRDIS plans provide is the following:

- `net = create_overlay(functionality[], type (optional))`
- `initialise(net, entry-points[])`
- `get_neighbors(net, number [optional])`
- `shutdown_overlay(net)`

Where the functionality parameter in the create_overlay operation is a set containing one or more of the following values: routing, membership, sampling, dissemination, or resourceLocation; and the type in the same operation is an optional parameter that allows the programmer to specify a specific overlay network (e.g., Kadmelia, Chord, HyParView, Scamp).

The other operations provide the fundamental and necessary interfaces for any overlay network, including a mechanism to allow a process to join an existing overlay by providing a list of processes currently in the system (usually called contacts); obtain a sample of other processes in the systems; and leave the overlay network.

The API also includes events that can be asynchronously triggered for the application (if the application registers handlers for these events), which include:

- `NeighborUp(Id)`
- `NeighborDown(Id)`

However not all overlays support this functionality.

Some types of overlay provide additional functionality which are exposed to other components of the system (e.g., communication protocols) and applications through additional API calls. These include, for instance, mechanisms to locate processes within an application-specific identifier space or to route messages to nodes given the application-specific identifier space being employed, and will be discussed in further detail on Deliverable D6.1.

In terms of additional functionality that can be provided at the Overlay Network level, the existence of a distributed certification platform may allow us to provide identity verification for processes when new neighbouring relationships are established at the overlay layer.

3.4.1.2. Communication Abstractions and APIs

Within the context of TaRDIS we plan to provide distributed protocols for a wide range of different communication primitives, going from simpler point-to-point primitives to more complex point-to-multipoint primitives, the latter usually operating on top of an overlay network. We aim also to provide mechanisms for inferring the appropriate type of overlay network to be used when the application developer identifies the type of point-to-multipoint communication primitive that they require.

Overall, all communication abstractions provided follow a similar API (presented below) where a method allows to create a communication channel, another allows to deactivate that

© 2023-2025 TaRDIS Consortium

channel, and then a set of abrasion-specific methods allow to interact with the communication channel (e.g., send a message, or register a topic of interest in a publish-subscribe channel).

Furthermore, we consider a set of desired properties that can be defined when a communication channel is instantiated, and that directly affect the implementation of the communication channel used at runtime. These include:

- **Reliability guarantees** — which specify if the communication channel operates in best effort, or if it provides more robust and fault-tolerant semantics (potentially with a notification to the application when the communication channel cannot guarantee delivery.
- **Order** — which specifies if the communication channel provides any guarantees over the ordering of messages (events) delivered across processes, these can be the following: node, fifo (first-in first-out), causal, or total.
- **Delivery semantics** — which specifies if delivery of events have specific guarantees, which can range from: none, at least once, at most once, and exactly once.
- **Security properties** — which allow to specify if the communication channel enforces any of the following security mechanisms (notice that some mechanisms might require the existence of an external distributed certificate authority): none, authentication, privacy, integrity (or combinations of the latter three).

In the following we illustrate the API proposed for the several classes of communication abstractions planned for TaRDIS, in all cases there is an additional mechanism that allows a component or application to register a handler to asynchronously process received events from a communication channel.

Point-to-Point

```
createDirectChannel(properties, destination, overlayNetwork (optional) ) -> channel
send(channel, msg)
registerHandler(channel, handler() )
tearDown(channel)
```

Broadcast

```
createBroadcastChannel(properties, overlayNetwork (optional) ) -> channel
send(channel, msg)
registerHandler(channel, handler() )
tearDown(channel)
```

Multicast

```
createMulticastChannel(properties, address, overlayNetwork (optional) ) -> channel
send(channel, msg)
registerHandler(channel, handler() )
tearDown(channel)
```

Publish-Subscribe

```
createPubSubChannel(properties, overlayNetwork (optional)) -> channel
publish(channel, event)
subscribe(channel, topic)
unsubscribe(channel, topic)
registerHandler(channel, handler() )
tearDown(channel)
```

Funded by the European Union

© 2023-2025 TaRDIS Consortium

3.4.1.3. Workflow-Based Communication API

Building upon the overlay network and messaging primitives described in Section 3.4.1.1, the TaRDIS toolbox provides a high-level abstraction for inter-process cooperation based on declared workflows. A workflow consists of an initial state and further states reachable by transitions that are represented by events replicated between the swarm participants. The workflow is driven forward by a group of nodes in the swarm, each playing a given role — each state transition is allocated to one role, the members of which are permitted to initiate it.

In accordance with the preliminary requirements described in D2.1, the replication and cooperation model adopted by the workflow API is fully asynchronous and permits each node to independently perform all actions that are allowed based on the partial knowledge that is locally available. Since eventual consistency is another requirement, TaRDIS offers verification tools that ascertain that a given workflow is a projection of the global swarm protocol onto a certain role — which guarantees that all non-failing participants will reach eventual consensus on the state progression performed by any execution of the workflow, without employing further coordination (see section 3.2.1.1 on communication analyses).

```
// definition language for specifying swarm protocols
const swarmProtocol = TaRDIS.defineSwarmProtocol(...)

// derivation of the workflow as seen by one particular participant role
const workflow = swarmProtocol.projectionFor("someRole")
const { State1, State2 } = workflow

// obtain a running instance of a workflow
const instance = await TaRDIS.findWorkflow(workflow, id)

// obtain a list of running instances of a workflow in a given set of states
const flows = await TaRDIS.findWorkflows(workflow, cutoff, [State1, State2])

// interrogate an instance's state
const state = instance.get()
if (state.is(State1)) {
	const cmd = state.commands() // commands currently enabled
	await cmd.command1(args…)    // invoke a command to emit an event
	const state2 = await.instance.next() // instance now has a new state
}

// consuming state update streams
for await (const state of instance) {
	// observe state changes, invoke commands, possibly break the loop
}
// `instance` is now destroyed, resources released

// event listener interface
instance.on('next', (state) => { ... })

// stop background updates and release associated resources
instance.destroy()
```


© 2023-2025 TaRDIS Consortium

The above API is presented in the TypeScript language as per the Actyx use case (Section 4.1), with provided elements marked in red. This API makes extensive use of the expressive features of the host language's type system to statically ensure correct usage of the workflow. A notable difference to linear channel usage models favoured by session types is that due to the dynamic swarm nature the state of a workflow can change at any time in the background, meaning that dynamic pattern matching is required for type-safe state access (in contrast to the statically known progression of channel types in linear channel models).

3.4.2. Decentralised Data Management and Replication APIs - T6.2

Different storage solutions can be produced to support the operation (and be integrated into) the TaRDIS toolbox. Storage solutions will differ between their architectural pattern (Section 3.4.2.1) and the guarantees provided to the application (Section 3.4.2.2): we discuss each one in turn, and then outline the proposed APIs (Section 3.4.2.3).

3.4.2.1. Architectural Patterns

- **Independent Subsystem**: This is the common approach now-a-days where a data management subsystem exists independently of the application, usually executing on dedicated infrastructure. This is the case, for instance, when an application interacts with a database that is centralised (such as a MySQL database) or distributed but logically centralised, or geo-distributed (such as Cassandra,[49] DynamoDB,[50] or CosmosDB[51]). While a geo-distributed/replicated database can be considered as fitting within the scope of TaRDIS, within the context of this architectural pattern it is more defensible to leverage on highly distributed data storage systems that have components both on the cloud and at edge locations (potentially extending towards the client applications).
- **Integrated Component**: This is an approach where the data management system is fully decentralised and emerges from the interactions between the different application components of a distributed system. In this context, each application process will run a local component of the data management system that interacts with similar components in other application processes of the system to collaboratively maintain the data management systems. Replication can be exploited within this context for both availability of data and data locality (which in network partition scenarios will directly impact availability). There are no obvious solutions now-a-days that completely fit within this model, so this is a venue of strong research and innovation for TaRDIS.

Notice that his architectural patterns are independent of other architectural choices such as the use of total or partial replication, static or dynamic replication/data placement, and others.

3.4.2.2. Provided Guarantees

Independently of the architectural pattern employed, each data management solution to be provided as part of the TaRDIS toolbox will provide a set of well defined guarantees at the interface level exposed to the programmer of applications. These include, but are not limited to:

[49] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922
[50] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12). Association for Computing Machinery, New York, NY, USA, 729–730. https://doi.org/10.1145/2213836.2213945
[51] https://learn.microsoft.com/en-us/azure/cosmos-db/introduction

Funded by the European Union

© 2023-2025 TaRDIS Consortium

- **Durability guarantees**: This type of guarantee captures in which conditions the effects of an operation that modifies the data managed by the system and that was confirmed by the system are ensured to remain durable (and visible) in the future of the system whatever happens. Can be defined in a quantified way (e.g., up to 3 faults).
- **Availability guarantees**: This type of guarantee captures conditions in which the data management system is able to return a result for any operation issued by an application. Can potentially be quantified, although unclear which metric could be employed.
- **Consistency guarantees**: These guarantees effectively restrict the data that can be returned by the data management system for an operation. These restrictions consider the history of the system and previous operations and their return values. Materializations of this includes causal consistency, read committed, etc
- **Data Integrity guarantees**: This type of guarantees captures invariants and other application-level restrictions for the data being managed by the data storage system.
- **Security**: Involves a set of properties that include: none (for no security), Integrity, Privacy, Access Control (or combinations of the three later).

3.4.2.3. Proposed APIs

```
create(dataSpace, properties)
create(dataSpace, tableID/keyspace, properties)
execute(dataSpace, tableID/keyspaceID, condition over objects, operation) -> result
delete(dataSpace, tableID/keyspace)
delete(dataSpace)
```

3.4.2.4. Existing and available solutions

The TaRDIS consortium has a large experience in the design and implementation of large-scale distributed storage systems, namely in geo-replicated data storage system offering causal+ consistently, such as ChainReaction[52] and C3[53], edge-systems providing configurable consistency guarantees such as the Engage[54] system, and even on the operation of decentralized storage systems such as Legion[55] and the Inter-Planetary File Systems (IPFS).[56]

---

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

[53] 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. https://doi.org/10.1109/NCA.2018.8548067

[54] 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. https://doi.org/10.1109/ICCCN54977.2022.9868846

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

[56] Costa, P.Á., Leitão, J., Psaras, Y. (2023). Studying the Workload of a Fully Decentralized Web3 System: IPFS. In: Patiño-Martínez, M., Paulo, J. (eds) Distributed Applications and Interoperable Systems. DAIS 2023. Lecture Notes in Computer Science, vol 13909. Springer, Cham. https://doi.org/10.1007/978-3-031-35260-7_2

Funded by the European Union

© 2023-2025 TaRDIS Consortium

3.4.3. Decentralised Monitoring and Reconfiguration APIs - T6.3

The TaRDIS toolbox provides primitives to support reconfiguration of the applications components in the runtime, as well as acquire the decentralised telemetry information from the deployed system, including information about the load and health conditions of different components. The toolbox will provide primitives to export fine-grained stored metrics to other parts of the toolbox, such as machine learning components that require time series of the stored metrics over some period of time.

This part of the toolbox will leverage the cloud-edge continuum, but also rely on open-source solutions as part of its core.

Though it is early to specifies APIs for these tasks, some of the proposed APIs may include:

- Decentralised monitoring:
```
metricsAggregate(type, namespace, range) -> metric
metricSeries(type, range) -> metric_range
```

- Reconfiguration management:
```
create(type, namespace, properties) -> result
delete(type, namespace) -> result
get(type, namespace) -> result
```

3.4.3.1. Existing and available solutions

For store and metrics manipulations, TaRDIS can leverage InfluxDB[57], Prometheus[58] as complete solutions, but we can also use some distributed databases if more complicated queries are required such as Cassandra[59], Scylladb[60] etc. and for metrics extraction, we can rely on metrics extracted from container platforms such as Docker[61], Containerd[62] etc. and to collect other metrics, traces OpenTelemetry[63] could be used. All projects are open-sourced.

57 https://www.influxdata.com/
58 https://prometheus.io/
59 Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922
60 https://www.scylladb.com/
61 https://www.docker.com/
62 https://containerd.io/
63 https://opentelemetry.io/

Funded by the European Union

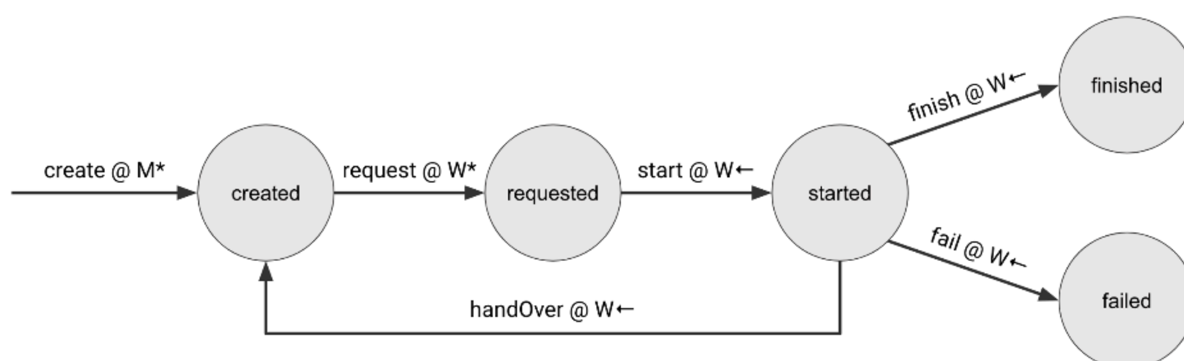
© 2023-2025 TaRDIS Consortium

4 OVERVIEW OF THE TaRDIS USE CASE APPLICATIONS

This section contains an overview of relevant parts of the use cases of the project partners Actyx (Section 4.1), EDP (Section 4.2), GMV (Section 4.3), and Telefónica (Section 4.4).

The purpose of this section is to outline how each use case plans to leverage the programming model and APIs provided by TaRDIS, and ensure that the use case requirements are aligned to what the TaRDIS toolkit will provide. The contents of this section complement Deliverable D2.2, which includes detailed use case and toolbox requirements, and has been developed concurrently with the present Deliverable D3.1.

4.1. Actyx



The Actyx use case involves the collaboration of applications according to the swarm protocol depicted above: a machine in a factory floor requests maintenance, which is then carried out by factory workers, while a manager overlooks the process.

It should be noted that in the pseudo-code presented below the swarm communication and data management primitives from WP6 are not explicitly visible, as we expect these to be used within the TaRDIS infrastructure to provide the more high-level facilities showcased below. For example, publishing events or invoking workflow commands will be communicated to other swarm peers using pub-sub primitives on an overlay network managed within TaRDIS.

4.1.1. Actyx App 0: Machine Requesting Maintenance

This simple application demonstrates how a machine operating on a factory floor asks for a maintenance task to be performed. It uses the TaRDIS communication primitives directly (e.g. `Tardis.publish(event)`), i.e. it presents an external application of these TaRDIS functions instead of the internal usage (shown in the next two applications). APIs provided by TaRDIS are highlighted in red, comments in green, and the language is TypeScript.

```
// workflow definition that includes which role is allowed to see
// what events
const maintenanceTask = TaRDIS.designSwarmProtocol(...)
const workflow = maintenanceTask.projectionFor("machine")

// we assume that there is an application which uses the code module
// below to request maintenance tasks when needed
const requestMaintenance = async (machineId, machineType, instructions) => {
```


© 2023-2025 TaRDIS Consortium

```
        const payload = { machineId, machineType, instructions }
        const id = uuid.v4()
        const event = maintenanceTask.events(id).create(payload)
        await TaRDIS.publish(event)
        return id
}

// this function can be used to obtain a future maintenance result
// using the `id` returned by an earlier call to
// `requestMaintenance`
const maintenanceResult = (id) => new Promise((resolve, reject) => {
        const task = await TaRDIS.findWorkflow(workflow, id)
        if ((await task.peekNext()).is(workflow.Initial)) {
                reject(new Error(`maintenance task ${id} not found`))
        }
        task.on('next', (state) => {
                if (state.is(workflow.Finished)||state.is(workflow.Failed)) {
                        resolve(state)
                        task.destroy()
                }
        })
})
```

While the requestMaintenance function directly emits the event that represents the request for maintenance (and hence stays outside the scope of behavioural verification), the maintenance result is computed by following the protocol embodied by the workflow object. This object describes the shape of the workflow, including the currently available actions, as seen by a particular participant role—the role of the "machine" in this case, as shown in the usage of the projectionFor method. The holistic workflow description from which the participant's view is projected must obey some well-formedness conditions to ensure that all foreseen roles will contribute to the workflow in a consistent fashion, reaching eventual consensus on how the execution proceeded. This is expressed in a unit test like the following:

```
// inspect the workflow for ambiguous messages or transitions
const result = TaRDIS.checkSwarmProtocol(maintenanceTask)
// this will print a detailed error report if the workflow has flaws
expect(result).toEqual({ type: 'OK' })
```

We demonstrate this verification step only here, but it applies equally to the other applications described below.

4.1.2. Actyx App 1: Maintenance Worker Tablet

This application is used by maintenance workers in a factory as their main planning and communication tool: they see open maintenance requests, select which one to work on next, and log their progress including completion or handover (e.g. due to their shift ending). Each maintenance request pertains to a specific machine in the factory and can be scheduled (for preventive or predictive maintenance) or immediate (in case the machine broke down).

The application is implemented in the TypeScript language, using the React UI framework. It interacts with the rest of the factory via the MaintenanceTask workflow: a machine, worker,

© 2023-2025 TaRDIS Consortium

manager, or an automated scheduling process creates a task and the app follows its progress and allows its user to interact with one MaintenanceTask at a time. It also can display an overview of a selection of currently open tasks with their current state details.

It should be noted that, during the execution of a task, the maintenance worker will also interact with the machine under maintenance, using a set of processes designed for this purpose (like transitioning the machine electronics and mechanics from production into maintenance mode, or the reverse process). The application will thus be part of more than one active process at a time; as each of these processes has a similar interplay with the React UI like the MaintenanceTask, we do not describe these aspects in greater detail. The important takeaway is that process execution cannot be exclusive, multiple processes can proceed in parallel. This ability is also required for updating a live overview of the eligible set of open maintenance tasks so the worker may select one.

In terms of security concerns (see also the related analysis of security properties in Section 3.2.3) the factory scenario offers only a limited amount of adversity: while some information shall not be available to some participants, the general setup is highly collaborative. It is sufficient to ensure the privacy of selected event payloads, we won't have to ensure the confidentiality of whether communication happened or who the participants were — the associated cost in terms of required network bandwidth and increased communication latency is not justifiable or even acceptable.

Pseudocode

The workflow as seen by the maintenance worker proceeds as follows:

```
begin:
    query list of all open maintenance tasks, filtered by capability (⇒ query the event store)
    sort list by priority, distance, ML inference, …
    select one task and observe its state (⇒ instantiate workflow and subscribe to state changes)
        reserve the task (⇒ invoke command to cause workflow transition)
        walk to the machine
        start the task (⇒ invoke command to cause workflow transition)
        perform maintenance
        finish, fail, or hand over the task (⇒ invoke command to cause workflow transition)
    ⇒ catch select: if the task is meanwhile reserved for someone else, stop working on it
loop begin
```

We first show the controller of this app that decides the UI states based on the model data (i.e. the maintenanceTask entities stored in TaRDIS). APIs provided by TaRDIS are highlighted in red, comments in green, and the language is TypeScript.

```
setUiState(Loading()) // populate UI state storage and trigger UI render

// workflow definition that includes which role is allowed to see what events
const maintenanceTask = TaRDIS.designSwarmProtocol(...)
const workflow = maintenanceTask.projectionFor("worker")

const capabilities = { ... } // machine types this worker knows

const prioModel = await TaRDIS.loadMLmodel(...) // task prioritisation

loop {
      // limit look-back to seven days for finding open tasks
```


© 2023-2025 TaRDIS Consortium

```
        const cutoff = new Date() - 7 * 24 * 60 * 60 * 1000
        // find tasks that are within the given set of states
        const tasks = await TaRDIS.findWorkflows(workflow, cutoff,
                [workflow.Created, workflow.Requested, workflow.Started])
        // treat the tasks as an array, which opts out of background updates
        tasks.filter((task) => task.get().payload.machineType in capabilities)
        tasks.sortBy((task) => prioModel.infer(task.get()))
        // yield tasks to UI for selection; tasks list is self-updating
        const selectionPromise = new Promise((resolve) =>
                setUiState(Overview(tasks, (id) => resolve(id)))
        const selected = await selectionPromise
        // switch to working on the selected task
        const task = tasks.find((t) => t.id === selected).clone()
        // release resources for keeping non-selected tasks updated
        // (TaRDIS should implement a caching strategy to optimise retrieval)
        tasks.destroy()
        await new Promise((resolve) => {
                // UI can make task progress or invoke closure to leave task
                setUiState(SingleTask(task, () => resolve()))
                // monitor task progress and switch back to overview when done
                task.on('next', (state) =>
                        state.is(workflow.Finished) || state.is(workflow.Failed)
                        ? resolve() : null
                )
        })
        task.destroy()
}
```

Note that the `tasks` list of tasks is a dynamic object that offers event emitter functionality. The same goes for the `task` entity object. The UI makes use of these facilities to update itself and offer the available choices to the end user as shown below on the example of the SingleTask view.

```
const SingleTaskView = (task, exit) => {
        const state = useWorkflow(task)
        if (state.is(workflow.Created)) {
                // state type is refined for Created, including commands
                const request = state.commands()?.request
                // commands get disabled once they have been invoked
                const disabled = request === null
                // barebones sketch of a UI with two buttons
                return <div>
                                <button onClick={exit}>Exit</button>
                                <button onClick={request}
                                        enabled={!disabled}>Request</button>
                        </div>
        } else if (state.is(workflow.Requested)) {
                return <div>...</div>
        } else if (state.is(workflow.Started)) {
                return <div>...</div>
```


© 2023-2025 TaRDIS Consortium

```
        } else {
            return <DoneView />
        }
    }
```

4.1.3. Actyx App 2: Manager Dashboard

This application is used by the production manager in a factory to oversee the planning and execution of maintenance tasks as described above. The manager can reassign or close tasks as well as fix mistakes made by maintenance workers (like leaving at the end of their shift without handing their current task over to the next shift). By labelling task executions as nominal or anomalous the manager provides ongoing training data for machine learning so that the application can identify anomalous tasks in real-time, allowing the manager to intervene and thus reduce overall machine downtime.

The application is implemented in the TypeScript language, using the React UI framework. Like the worker app it interacts with the rest of the factory via the MaintenanceTask process. In addition to task overview and single task interaction, it also monitors the completion of maintenance tasks, labels them as nominal or anomalous via heuristics or user interaction, and feeds the thusly labelled task execution traces to an ML process. The continuously trained ML model is then used to monitor ongoing task execution for anomalies and alert the user of each one found.

This application demonstrates that historical data needs to be stored such that the manager dashboard can access them: execution traces need to be extracted to train ML models. Besides this TaRDIS-related usage, the execution traces are also needed by the factory management in order to perform cost evaluation, identify optimisation potential, and if necessary perform audits.

The security concerns exhibited by this application match those of App 1 above.

Pseudocode

The manager dashboard application offers a single screen that presents useful information and allows some actions (like stopping a task that was completed but where the worker forgot to register that fact). Conceptually it keeps updating the screen in an endless loop like this:

initialise two lists as empty: closedTasks and openTasks
begin:
query list of IDs of all maintenance tasks that were recently closed (⇒ query the event store)
remove task IDs that are already on closedTasks from this list
run ML inference to get nominal/anomalous status for each task's whole history (⇒ use ML API)
add heuristic labels to tasks (noisy) based on production expert inputs
append task IDs to closedTasks (to prevent them from being processed again)
feed labelled tasks into federated learning service (⇒ use ML API)

query list of IDs of all open maintenance tasks (⇒ query the event store)
run ML inference to get nominal/anomalous status for each task's whole history (⇒ use ML API)
compute current workflow state for each task (⇒ instantiate workflow state)
replace openTasks with list of tuples (task ID, anomaly status, workflow state)

display closedTasks, openTasks, and FL training status & metrics in suitable UI,
allowing manager to *perform state updates* like a worker (⇒ run workflow command)
loop **begin** (after suitable delay)

© 2023-2025 TaRDIS Consortium

As the UI uses the same primitives as in App 1, we concentrate on the controller. This one consists of two loops that continually update the UI state: one event-driven by the completion of tasks, the other driven by a timer to assess the progress of ongoing tasks at all times (i.e. also when no new events are produced).

```
setUiState(Loading()) // populate UI state storage and trigger UI render

// workflow definition that includes which role is allowed to see what events
const maintenanceTask = TaRDIS.designSwarmProtocol(...)
const workflow = maintenanceTask.projectionFor("manager")

const taskModel = await TaRDIS.loadMLmodel(...) // task anomaly detection

// subscribe to the dynamically changing set of ongoing tasks
const tasks = TaRDIS.findWorkflows(workflow, cutoff,
      [workflow.Created, workflow.Requested, workflow.Started])

// upon completion of a task, run inference (for UI display) and feed into
// machine learning infrastructure with (noisy) label for model improvement
tasks.on('remove', (task) => {
      const eventLog = task.history()
      const score = taskModel.infer(eventLog)
      const noisyLabel = computeHeuristicLabel(eventLog)
      taskModel.addTrainingRecord(eventLog, noisyLabel)
      addUiClosedTask(task, score, noisyLabel)
})

// ML background process can provide state updates for the UI as well
taskModel.on('update', (state) => setUiMLstate(state))

// regularly inspect ongoing tasks to see whether they are still on track
setInterval(() => {
      const tasksForUi = tasks.map((task) => {
            const eventLog = task.history()
            eventLog.push(new Date()) // inference needs to know current
            time
            const score = taskModel.infer(eventLog)
            return [task, score]
      })
      setUiOpenTasks(tasksForUi)
}, 30_000) // update ongoing task inference every 30sec
```

It should be noted that the `loadMLmodel` function receives as arguments a definition of the model design, the function that converts event histories into ML inputs, and any other parameters that are needed for starting a federated learning agent that can asynchronously perform incremental training improvements and exchange improved models with swarm peers. We expect the number of generated training records to be of the order of 100–1000 per day, with mostly stable workflow conditions for weeks at a time.

© 2023-2025 TaRDIS Consortium

4.1.4. Actyx App 3: Real-Time Monitoring

This application is in a very early design stage. The goal is to develop a perimeter TaRDIS service that monitors a running system for specific combinations of events — for instance, whether a machine (or group of machines) emits multiple requests for maintenance in a short time frame. Manually writing programs that identify complex combinations of events can be error-prone and time-consuming, hence we plan to leverage the join pattern matching API provided by TaRDIS (Section 3.2.1.3) (in collaboration with DTU) to tackle the problem and reduce development time.

4.2. EDP

4.2.1. Background and general objective of the Energy use case

An Energy community is a network of consumers and producers, in a delimited geographical region, who collectively manage and share energy resources.

For electricity, the connection between producers and consumers is established through physical cables with connection points called nodes, establishing a grid. At these grid nodes, voltage levels, frequency and phase need to be always stable within the limits - then one can say that the grid is balanced - and this happens for both direct-current (DC) and alternate-current (AC) grids.

Having the grid balanced is a sufficient condition to say that production meets consumption. Nowadays, due to Renewables with no primary energy costs associated, one can also work the other way around by, for instance, deferring consumption in time.

Geography and demographics drive electric grid sizing. Technically, cables and nodes should support the power flow but as distance from generation increases, losses in the cables become not neglectable. So, having production nearby demand is way better than energy transportation. In this context, Energy communities (i.e., localised ensembles of energy producers and consumers that are interconnected) play an important role.

Although there are several Energy communities already in place, centralization on the role of the community aggregator or the distribution system operator (DSO) is a limitation to citizens' energy trading participation, since registration, day-ahead forecast of production or faults need to have human intervention, due to regulation.

If each peer is able to connect within its community and make automatic agreements all the processes would be optimized, it could run 24h/7d with reduced human intervention and costs would be reduced also. This is the perfect ground for applying the TaRDIS toolbox.

4.2.2. Energy use case components and objectives

The interactions between peers within a community take the major part of the specifications, while the format of the message stands as the second most important. The former will be described using communication diagrams, depicting the interaction between prosumers in both the role of a consumer and producer, and the community orchestrator -responsible for the external interactions with other communities' orchestrators and DSO. The latter is statically described, remaining simple and stable, and will be used in different contexts, with different purposes.

There are three different messages with different fields:

1. `Post` - This message signifies the posting of a new offer to our market, and thus it has an **amount** of energy in kWh that the supplier is offering, it has a **price** per kWh

© 2023-2025 TaRDIS Consortium

offered, it has a **time** which is the period in which the supplier is claiming to produce the energy, and it has a **location**, so that offers from very far away can be dismissed, and it also has an **id**, so we can identify this posting in other messages.
2. `Bid` - This message has an **amount** of energy we want to buy, it has a **price** we suggest to buy the energy at, it has a **postId**, to know which post message this bid refers to, and it has its own **id**.
3. `AcceptBid` - This message simply has the **id** of the Bid that it attempts to accept.

The objectives are by precedence:

1. Maximize the use of Renewable energy inside an Energy community;
2. Have all consumers within the community with Energy supply guarantee;
3. Reduce the number of messages between peers within the communication network;
4. Use same code and equipment for all actors;
5. Have a system that can, in the future, incorporate intraday market bids.

4.2.3. Working Principles

For the system to operate two stages are needed, the agreement on supply-consume planning (ex-ante working mode) and the effective exchange of energy.

Demand should lead the process as human needs are based on energy consumption. Demand-side management is not yet considered.

As in a market, a match occurs when the supply cost and demand offer are met. In this case, an energy request from each consumer triggers bids from energy suppliers and each consumer can choose its suppliers, for the period, based on its own criteria (price, source, energy peak, etc.)

4.2.4. Requirements to TaRDIS

This use case has three crucial requirements from the TaRDIS runtime support.

1. Data storage mechanisms in the maintenance of the log of transactions and standing energy offers and energy requests. The properties required in this case are durability, authentication and non-repudiation (accountability).
2. Confidentiality and integrity of the matching process needs to be ensured by both analyzing the specified workflows and protecting communication channels using appropriate security protocols.
3. The decentralised machine learning mechanisms are relevant towards this use case to allow the system to predict the ability of a specific community to serve a request for energy, or if that request should be immediately routed towards the grid. These predictions must be performed while ensuring the privacy of prosumers regarding their energy production and consumption over time. This will take advantage of decentralised and privacy-enhanced techniques developed within WP5.

4.2.5. Scenarios

For the energy use case, we plan to have at least six scenarios, with two considered as main scenarios. The sequence chart, along with event-based pseudo-code, represents these main scenarios. The goal is to specify the base dependencies and minimum configuration for functionality.

Of the remaining four scenarios, two address specific situations in the energy community when the energy is not balanced. The first situation involves a deficit of energy, while the second deals with a surplus. The final two scenarios describe the “Run” mode, where we

© 2023-2025 TaRDIS Consortium

have already prepared the next hour time slot and entered into operation. In this mode, we can either have normal operation or operate with faults.

In the following diagrams, we describe a set of scenarios for this use case, considering one-hour periods. The scenarios are divided into two groups, one regarding the planning work “Ex-ante” where all peers agree on Energy production and distribution just before the new period starts and another group where real-time (operation) work is described as “Run”.

This extended DCR graphs-based language outlined in Section 2.2.1 is used here to define the use case workflows by means of events and event relations that are statically defined but whose instances are dynamically created on top of a communication API that supports the local execution of events and propagation of state to other members of the swarm.

DCR graphs provide a high-level declarative programming model that allows easy specification and reconfiguration of workflows. Event declarations also define the associated data items and their data dependencies; thus, the data storage model can be defined implicitly in the workflow.

The language includes the capability of querying the state of other input events or other data elements (computation events). A connection to the ML APIs is essential here to guide the matchmaking process in this use case. The semantics of query expressions used in the DCR graphs can be powered by ML mechanisms and APIs.

External applications can register to execute events of the workflow having the appropriate emitter role. Said applications can also register to listen to events that they are set to receive and that they depend on to proceed. The enableness of an event, which declares that the workflow is in a state that allows the execution of an event, should be established locally based on the notice of execution of events in the system, propagated/replicated through the decentralized system (swarm).

4.2.5.1. Scenario 1 - Ex-ante working Energy generation forecast for the next hour

We outline the process for obtaining the energy generation forecast for the next hour. The scenario describes the stage when the Community Orchestrator requests the producers how much energy they will produce in the next period, calculating the total available. The process then moves to the collection of feedback, concluding with the update of the Community Orchestrator's records of generation.

Funded by the European Union

© 2023-2025 TaRDIS Consortium

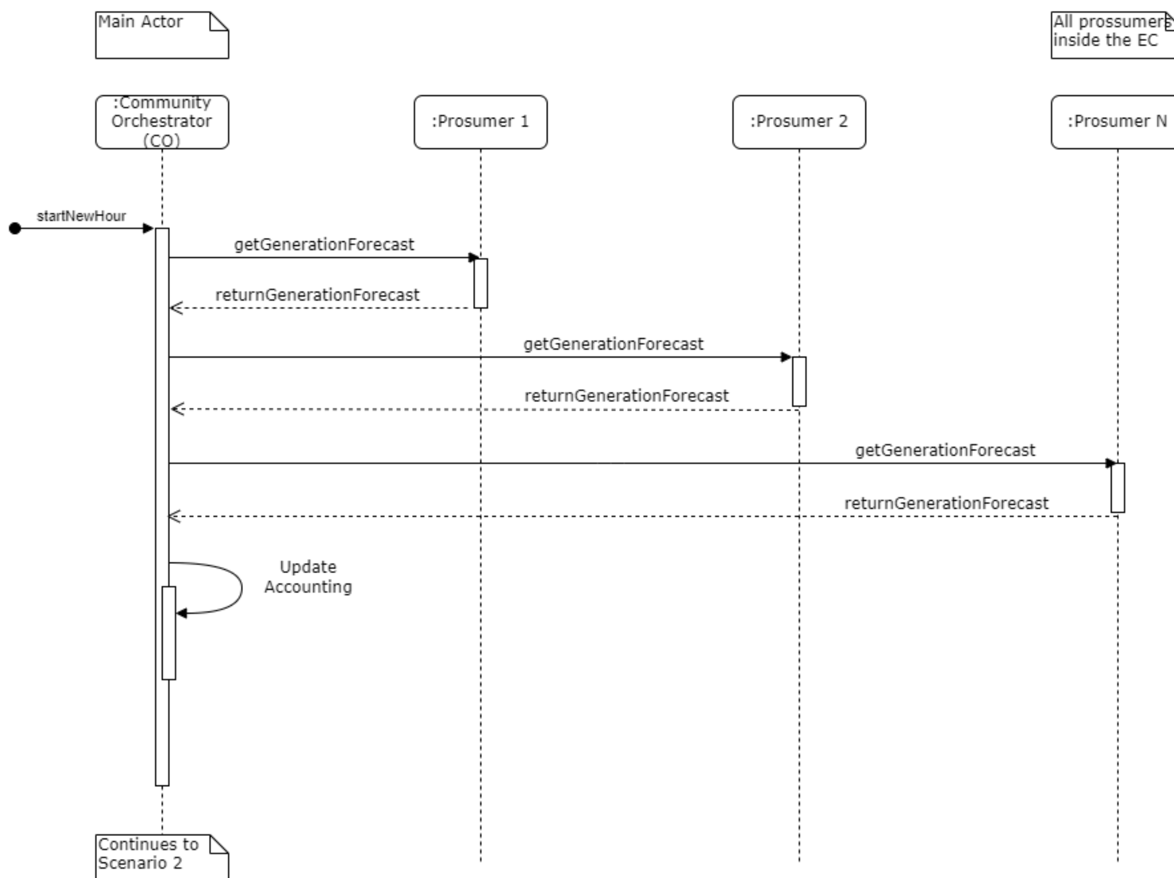


*Figure from D2.2 "S1 sequence diagram"

DCR/ReGraDa specification code

We now depict this sequence chart using DCR graphs. We use the roles Prosumer and Orchestrator.

```
rpf: RequestProductionForecast: Orchestrator(cid) -> Prosumer(cid,*)
gpf1: GenerateProductionForecast: Prosumer(cid, 1) -> Orchestrator(cid)  // producer 1
gpf2: GenerateProductionForecast: Prosumer(cid, 2) -> Orchestrator(cid)  // producer 2
gpf3: GenerateProductionForecast: Prosumer(cid, 3) -> Orchestrator(cid)  // producer 3
acc: AccountingCommunity[?:{kw}]: Orchestrator(cid) -> Orchestrator(cid)
;
rpf *-> gpf1
rpf *-> gpf2
rpf *-> gpf3
rpf *-> acc
gpf1 -><> acc (with timeout)
gpf2 -><> acc (with timeout)
gpf3 -><> acc (with timeout)
```

This depicts the instances of data elements (events) representing the community in the sequence chart, different instances of the same event (GenerateProductionForecast) exist per prosumer involved (`gpf1`, `gpf2`, `gpf3`). The explicit creation of these elements will be made at the end of this section.

© 2023-2025 TaRDIS Consortium

In this definition, we can observe that the event `rpf` will make the three instances `gpf1`, `gpf2`, and `gpf3` pending. That means that the prosumers will have to execute them. It also makes the event `acc` pending. Then, a milestone relation flows in the reverse order from all `gpf` events to the `acc` event in that community, which means that `acc` will not be executed while the former are pending (not been executed since `rpf` was executed.)

Pseudo-code Scenario 1

We now present the projected behaviour of each one of the roles in the different respective applications. We use an event-based pseudo-code that uses the TaRDIS API.

Orchestrator app for Generation Forecast:

```
State:
        i: iteration (logical clock)
        broadcastChannel: community broadcast channel identifier

Upon init do:
        broadcastChannel<-TaRDIS.createBroadcastChannel(props) // props are a set of properties
                                                               // that specify the type of
                                                               // properties desired from the
                                                               // broadcast channel
        TaRDIS.create("EnergyCommunityData", "GenerationForecasts",
                        props) // props are a set of properties related with the storage abstraction
                               // being requested
        i <- 1
        BroadcastPeriod <- 60 min
        ResponseTimeOut <- 10 min
        setup periodic timer ("HourlyForecastUpdate", BroadcastPeriod)
        TaRDIS.send(broadcastChannel, RequestProductionForecast(i))
        setup timer ("TimeOut", ResponseTimeOut)

Upon timer ("HourlyForecastUpdate") do:
        i <- i + 1
        TaRDIS.send(broadcastChannel, RequestProductionForecast(i))
        setup timer (TimeOut) (ResponseTimeOut)

Upon Receive GenerateProductionForecast (v, i') from Prosumer(p) do:
        // v: forecast value; i': period considered; p: prosumer's id
        if(i == i') then
          CollectionData <- CollectionData U (p,v)
          TaRDIS.execute("EnergyCommunityData", "GenerationForecasts", STORE(p,v,i) )

Upon timer ("TimeOut"):
        // check available information
        listOfForecasts <-
                TaRDIS.execute("EnergyCommunityData", "GenerationForecasts",
                                "id = i", RETRIEVE * )
        globalGenerationForecast <- computeGlobalHourlyForecast(listOfForecasts)
        TaRDIS.execute("EnergyCommunityData", "GenerationForecasts",
```


© 2023-2025 TaRDIS Consortium

```
            STORE(globalGenerationForecast, i))
```

Prosumer app for Generation Forecast:

```
Upon Receive RequestProductionForecast(i) from Orchestrator do:
        v <- TaRDIS.ML("computeGenerationForecastNextTimePeriod", i)
        Send GenerateProductionForecast(v, i) to orch
```

4.2.5.2. Scenario 2 - Ex-ante working Energy consumption forecast for the next hour

Consumers take scenario 1 as a tacit signal from the orchestrator that the new period is about to start and broadcast their consumption, then negotiation between producers and consumers starts and, using its own criteria, each consumer books production from producers. The model can optionally use the price. The orchestrator can now totalize the community's internal consumption.

In the second scenario, we outline the process of acquiring the energy consumption forecast for the next hour. The process initiates from the consumer side, where a consumer, considered as a prosumer unable to meet their own energy requirements for this timeslot, begins by requesting available energy from their prosumer peers. Subsequently, the consumer seeks energy offers from the prosumer peers (producers), proceeding to the third step of consumer selection of the best offer. The process concludes with the communication of information to the selected producer and the community orchestrator for the updating of accounting records.

Funded by the European Union

© 2023-2025 TaRDIS Consortium

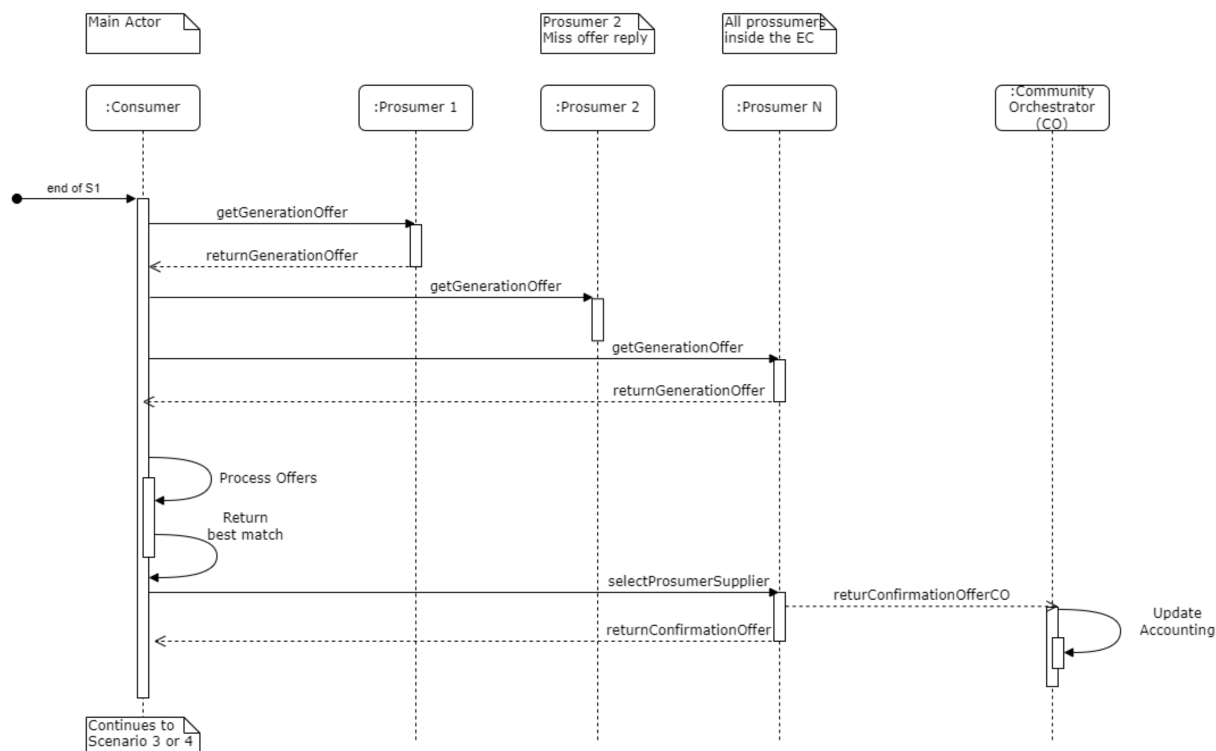


*Figure from D2.2 "S2 sequence diagram"

DCR/ReGraDa specification code

Once again we depict the scenario with DCR/ReGraDa graphs.

```
r1:EnergyRequest[?:{kw, t, prosumer}] : Prosumer(cid, pid1) -> Prosumer(cid,*),
Orchestrator(cid)
o2:EnergyOffer[?:{kw, t, pid}]: Prosumer(cid, pid2) -> Prosumer(cid,pid1)
o3:EnergyOffer[?:{kw, t, pid}]: Prosumer(cid, pid3) -> Prosumer(cid,pid1)
o4:EnergyOffer[?:{kw, t, pid}]: Prosumer(cid, pid4) -> Prosumer(cid,pid1)
s:SelectionOfSupplier[?:{consumer,producer}]: Prosumer(cid, pid1) -> Prosumer(cid,pid2)
c:Confirmation[?:{}]: Prosumer(cid,pid2) -> Orchestrator(cid)
acc: AccountingCommunity: Orchestrator(cid) -> Orchestrator(cid)
;
r1 *-> acc
s -->% r1
s ->% o2
s ->% o3
s ->% o4
s *-> c
c -->* acc
```

This depicts the instances of data elements (events) representing the community in the sequence chart, different instances of the same event (`EnergyOffer`) exist per prosumer (producer) involved (o1, o2, o3). The explicit creation of these elements will be made at the end of this section.

© 2023-2025 TaRDIS Consortium

In this definition, we can observe that the event r1 will make pending the accounting event in the community and that the event represents the explicit selection of a match. The three instances o1, o2, and o3 are excluded when the selection is made. The selection also makes the confirmation mandatory. The process also prevents the acc event from happening before the confirmation is executed.

Pseudo-code Scenario 2

The projected behaviour of each role can be therefore illustrated by the following (event-based) pseudo-code.

Consumer(c) app for getting supply offers:

```
State:
        i: iteration (logical clock)
        broadcastChannel: community broadcast channel identifier
        directChannels: map (by process) of direct communication channels

Upon init do:
        TaRDIS.create("EnergyCommunityData", "SupplyOffers",
                        props)   // props are a set of desired properties for the data storage
                                 //  abstraction and data collection being accessed.
        // current period (iteration i) retrieved from community context storage
        i <- TaRDIS.execute("EnergyCommunityData", "GenerationForecasts", RETRIEVE MAX(i) )
        ResponseTimeOut <- 10 min
        broadcastChannel <- TaRDIS.createChannel(broadcast,
                                        props) // attach to the broadcast channel,
                                               // the props is a set of properties
                                               // specified for this communication
                                               // channel
        TaRDIS.send(broadcastChannel, EnergyRequest(i) )
        setup timer ("TimeOut", ResponseTimeOut)

Upon Receive broadcast RequestProductionForecast(i') //This message was sent in scenario 1
        i <- i'
        TaRDIS.send(broadcastChannel, EnergyRequest(i))
        setup timer ("TimeOut", ResponseTimeOut)

Upon Receive EnergyOffer(v, i') from Prosumer(p) do:
        if(i == i') then
                CollectionData <- CollectionData U (p,v)
                TaRDIS.execute("EnergyCommunityData", "SupplyOffers", STORE (p,v,i,c,RECEIVED) )

Upon timer ("TimeOut"):
        // check available offers - assumes inherently getting offers only for myself
        listOfOffers <- TaRDIS.execute("EnergyCommunityData", "SupplyOffers", "id = i", RETRIEVE * )

        if (listOfOffers is empty) then:
                supply from EDP grid // exiting out of the TaRDIS context
        else:
                // internal function for picking the best match
```


© 2023-2025 TaRDIS Consortium

```
        selectedProducer <- TaRDIS.ML("calculateBestMatch", listOfOffers)
        // create direct communication link to selected producer
        if directChannels[selectedProducer] == NIL
                directChannel <- TaRDIS.createDirectChannel(properties,
                                                             SelectionOfSupplier)
        TaRDIS.send(directChannel, SelectionOfSupplier(v,i))

Upon Shutdown do:
        TaRDIS.tearDown(broadcastChannel)
        for(dc in: directChannels):
                TaRDIS.tearDown(dc)
```

Prosummer(p) app for Generation Forecast:

```
Upon init do:
        TaRDIS.create("EnergyCommunityData", "SupplyOffers", props //props are a set of desired
properties for the data storage abstraction and data collection being accessed.


Upon Receive EnergyRequest(v,i) from consumer(c) do:
        availableGenerationValue <- TaRDIS.ML("computeGenerationForecastNextTimePeriod", i)
                if(v<=availableGenerationValue) then:
                 TaRDIS.send(consumer, EnergyOffer(v,i))
                 TaRDIS.execute("EnergyCommunityData","SupplyOffers", STORE (p,v,i,c,OFFERED))
                 setup timer("ConfirmationTimeOut(p,v,i,c,OFFERED)", ConfirmationTimeOut)

Upon ConfirmationTimeOut(p,v,i,c,OFFERED) do:
        TaRDIS.execute("EnergyCommunityData","SupplyOffers", STORE (p,v,i,c,UNFULFILLED))

Upon Receive SelectionOfSupplier(v,i) from Prosumer(c):
        TaRDIS.send(c.directChannel, Confirmation(v,i,c))
        TaRDIS.execute("EnergyCommunityData","SupplyOffers", STORE (p,v,i,c,CONFIRMED))
```

Community Orchestrator(O) App for Generation Forecast:

```
Upon Receive Confirmation(v,i,c) from Prosumer(p):
        // internal functions for accounting
        executeAccounting(Confirmation(v,i,c),p)
```

4.2.5.3. Consolidated DCR/ReGraDa code for the two previous scenarios

Finally, we gather the above examples in a complete DCR/ReGraDa program that depicts both scenarios as well as the bootstrap process. In this final example, we use the `spawn` relation to create the communities and their prosumers, thus instantiating the different events that correspond to the different elements of the graphs above. With relation to the previous excerpts, this example contains an event `consume` and an event `replyConsume` that are needed to correctly pair the events and allow a prosumer to issue multiple requests and offers.

```
createCommunity[?:{ cid }]
```


© 2023-2025 TaRDIS Consortium

```
;
createCommunity ->> {
  ci: CommunityInfo[{@trigger.cid}]: * -> *
  cp:createProsumer[?:{pid}]: Orchestrator(@trigger.cid) -> *
  acc:accountingCommunity[?:{kw}]: Orchestrator(@trigger.cid) -> Orchestrator(@trigger.cid)
  rpf:requestProductionForecast: Orchestrator(cid) -> Prosumer(@trigger.cid,*)
  ;
  rpf *-> acc
  cp ->> {
    pi: ProsumerInfo[{pid = @trigger.pid}]
    pf: generateProductionForecast[?:{}]:
      Prosumer(ci.cid, @trigger.pid) -> Orchestrator(ci.cid)
    c: consume[?:{kw}] :
      Prosumer(ci.cid, @trigger.pid) -> Prosumer(ci.cid, @trigger.pid)
    p: replyConsume[?:{kw, consumer}]: Prosumer(ci.cid, @trigger.pid) ->
                                                  Prosumer(ci.cid, consumer )
    ;
    rpf *-> pf
    pf -><> acc (with timeout)
    consume ->> {
      r:!EnergyRequest[{kw,t}]:
        Prosumer(c.cid, @trigger.pid) -> Prosumer(c.cid,*), Orchestrator(ci.cid)
        s: selectionOfSupplier[?:{producer}]:
          Prosumer(c.cid, @trigger.pid) -> Prosumer(ci.cid, producer)
       ;
       r *-> acc
       s ->% r
       s ->% o:EnergyOffer    if(s.producer == o.producer)
       s ->% c:Confirmation   if(s.producer != c.produce and c.consumer == pi.pid)
    }
    replyConsume -> {
       o:EnergyOffer[{consumer = @trigger.consumer, producer = pi.pid}]:
         Prosumer(c.cid, pi.pid) -> Prosumer(c.cid, @trigger.consumer)
       c:Confirmation[{consumer = @trigger.consumer, producer = pi.pid}}]:
         Prosumer(ci.cid,pi.pid) -> Prosumer(c.cid,@trigger.consumer), Orchestrator(c.icid)
       ;
       c ->% c
       o ->* c
    }
  }
}
```

This code is just a sketch of what a process capturing the behaviour of the prosumers could look like.

4.2.5.4. More scenarios

The remaining 4 scenarios are described shortly in the next items:

- Scenario 3 - Ex-ante working with total energy consumption forecasted for next hour and balance of deficit.
    - After Scenario 1 and 2, in this scenario where the community is unbalanced and requires energy from external sources, the community orchestrator takes on the role of a consumer and requests energy from other community orchestrators. Two possible situations arise from this: either the remaining energy needs are fulfilled by the other community orchestrators, or if not, the community orchestrator requests the remaining deficit from the grid (DSO)

- Scenario 4 - Ex-ante working with total energy consumption forecasted for next hour and balance of surplus.

© 2023-2025 TaRDIS Consortium

  - After Scenario 1 and 2, it is possible that our community has surplus of energy being also unbalanced, the community orchestrator takes on the role of a producer and offers energy to other requesting community orchestrators Scenario 3. Two possible situations arise from this: either the surplus of energy is fulfilled by the other requesting community orchestrators, or if not, the community orchestrator injects the remaining surplus to the grid (DSO).

- Scenario 5 – Running in Normal operational mode.
  - In the normal operation mode, following Scenario 2, the consumer receives the energy offer from the selected producer, acknowledges this offer, informs the community orchestrator, and ask the producer to initiate the energy transaction. Finally, the consumer notifies both the producer and orchestrator of the total energy consumed, allowing them to verify and update the records.

- Scenario 6 – Running with faults.
  - In this final scenario, two types of faults may occur, either from the supply side or the demand side. In the case of a supply fault, the consumer notifies the community orchestrator of the issue. The orchestrator resolves the energy problem using information from Scenario 1. If it progresses to Scenario 3, the grid will always fulfil the consumer's needs. The orchestrator then tags the producer as not meeting the agreement. If maintenance is required, it will be carried out by a human, and if the fault is recurring, the producer may be excluded from the pool. In the case of a demand fault, the producer notifies the community orchestrator, ceases production if unable to do it, starts injecting into the grid, and the consumer is not alerted.

While in the presentation of these use case scenarios we have relied on the notion of a centralised orchestrator, for both simplifying the exposition of these scenarios and to ensure that these closely maps to the baseline of this use case; in the context of TaRDIS the orchestrator will be materialised in a decentralised way, where different processes across presumers can coordinate between them to provide this functionality. The distribution (and decentralisation) of the orchestrator can be achieved while ensuring that the API (and events) processes are the same as the ones modelled here.

4.3. GMV

The GVM use case application will be used by space engineers for designing distributed Orbit Determination and Time Synchronization (ODTS) algorithms for a constellation of satellites, to study and optimize their performance.

The GMV use case applications will leverage the following TaRDIS APIs providing a perimeter service:

- AI/ML programming primitives (Section 3.3.1): will be used to define ML models for the Orbit Determination and Time Synchronization algorithms
- Lightweight and energy-efficient ML library (Section 3.3.3): will be used to minimize computational load of the ML models/tasks. This is particularly important in view of a future implementation of the models on a satellite on-board computer.

The GMV use case will produce two applications:

- a reference sequential simulation of the decentralised orbit determination (Section 4.3.1), written mainly using Matlab and Python. This application will use the TaRDIS

© 2023-2025 TaRDIS Consortium

AI/ML APIs in combination with well-established modelling and simulation techniques adopted in the space industry; and
- a distributed simulation (Section 4.3.2) that will additionally leverage the TaRDIS APIs for communication and distribution, to simulate a swarm of satellites performing decentralized ODTS by modelling each satellite as an independent distributed application instance.

4.3.1. Sequential Orbit Simulation Application

An example of a very high level pseudo-code of a simulation is outlined below.

```
settings = setParameters() // establish parameters settings for ODTS simulation
orbital_data = getOrbitalData(orbit_data_file) // get satellites orbital data
connections = islScheduling(orbital_data) // inter-sat connections over time
for block_number = 1 : N // iterate over blocks
       for t_slot = 1 : n // iterate over time slots
              for sv_number = 1 : n_sats
                     obs = getMeas(connections(t_slot),
                                   orbital_data,
                                   settings) // simulate measurements
                     state_update = odts(state, obs) // performing ODTS
                     state = state_update
              end
       end
end
```

TaRDIS APIs will be used within the ‘`odts`’ function mentioned in the pseudo-code.

4.3.2. Distributed Simulation Application Based on PTB-FLA

The pseudo-code of a distributed PTB-FLA based simulation is outlined below:

```
ptb = PtbFla(no_nodes, node_id) // create the object ptb (testbed start up)
settings = setParameters() // parameters settings for the ODTS simulation
orbital_data = getOrbitalData(orbit_data_file) // get satellites orbital data
connections = islScheduling(orbital_data) // inter-sat connections over time
for block_number = 1 : N // iterate over blocks
       for t_slot = 1 : n // iterate over time slots
                     obs = ptb.getMeas(node_id, connections(t_slot),
                                       orbital_data,
                                       settings) // simulate measurements
                     state_update = odts(state, obs) // performing ODTS
                     state = state_update
       end
end
del ptb // delete the object ptb (testbed shutdown)
```

Obviously, transforming the original pseudocode for the sequential (centralised) orbit simulation (Section 4.3.1) into the pseudocode for the distributed PTB-FLA simulation (Section 3.3.1.1) is rather straightforward and requires just two modifications:

1. create the object `ptb` at the beginning of the simulation and delete it at the end,

Funded by the European Union

© 2023-2025 TaRDIS Consortium

2. replace the call to the function `getMeas` with the call to the function `getMeas` on the object `ptb` (denoted as `ptb.getMeas`); note that the latter function requires the additional argument `node_id`.

Although the two pseudo code samples look almost identical, there is one major difference between them: in the first pseudocode the variables `obs`, `state_update`, and `state` contain data for the *complete system* (i.e., the satellite constellation), whereas in the second pseudocode they contain data only for the *local node of the system* (i.e., the individual satellite being simulated by the local node) whose PTB-FLA ID is `node_id`. The big difference between the two pseudo-codes is that with PTB-FLA the loop over the number of satellites is avoided, thus allowing to simulate the ODTS algorithm for each satellite in parallel (and potentially in a distributed system) and not in a sequential mode.

4.3.3. Roadmap

GMV and UNS plan to try to implement the second pseudocode later during the TaRDIS project; tentatively GMV will translate their sequential simulation from Matlab to Python, UNS will implement the new function `getMeas`, and the resulting PTB-FLA based distributed simulation will be tested as a group of processes on a single computer.

4.4. Telefónica

Telefónica's use case is a Federated-Learning-as-a-Service (FLaaS) platform designed to support various types of applications running on end-user or edge devices. At the moment, this application also requires a centralized back-end server to coordinate the construction of the FL global model.

Main languages used for the service are Python for the backend server and Java for the Android OS-based applications using FLaaS middleware.

TaRDIS primitives, APIs and tools that FLaaS will take advantage of are described in the following subsections.

4.4.1. Analyses for security (T4.3, Section 3.2.3)

This analysis is expected to be useful for the secure communications between the various entities of FLaaS, i.e., backend server, intermediate / super-nodes, end-user or edge devices. The use of this analyses' outputs will be useful to provide guarantees about the entities participating in the communications between such entities. The pseudo-code below outlines how secure connections are expected to be established.

```
(NetId, Conn) = check_connectivity  # prompts the network (swarm)
                                    # for connectivity status
```

The output consists of:

- `NetId` is the array of identities of the all the network nodes (e.g., end-users, super/compute nodes, aggregator),
- `Conn` is the array of communications links between all the network nodes (e.g., 0 or 1)

```
Sec_conn = establish.sec.connections(NetId, TrFlag, Conn,
                                     security_parameters,
```


© 2023-2025 TaRDIS Consortium

```
                                        Thresholds) # establishes secure
                                                    # connections in the network
                                                    # according to predefined
                                                    # rules, i.e., encryption
                                                    # protocols to be applied.
```

The arguments are:

- NetId as defined above
- TrFlag is an array of the trust level of each node
- Conn as defined above
- security_parameters: the predefined rules for security among the nodes
- threshholds: an array with acceptable thresholds related to security_parameters

The functionalities above can be either part of an API or built-in within the FLaaS middleware.

4.4.2. AI/ML programming primitives (T5.1, Section 3.3)

The output of this task is expected to be used for providing:

1. examples of readily deployable code for distributed AI applications (which can be of use while building the new version of FLaaS);
2. an optimization-based module for configuring execution of FL tasks on end-user and edge devices, by taking into account constraints at hand such as device types and on-device system parameters (computation power, memory, communication), FLaaS network topology, etc.;
3. a way to help deployment of FLaaS tasks on devices, by taking into account the different execution environments: swarms (e.g., for distributed optimisation/P2P federated learning for the organization of FLaaS resources in a hierarchical fashion) and device-edge-cloud environments (e.g., for using split learning for the FLaaS tasks).

FLaaS will make use of the runtime environment for FL algorithms, PTB FLA API, as described in the related Section 3.3.1.1 above. In particular, it will call the constructor (PtbFla) to initiate an instance and then the fl_centralized function (for FL execution between the server and the end-users). It can possibly make use of the fl_decentralized (in the case of cross-app on-device FL). Finally, it calls the destructor to end the instance (PtbFla()).

4.4.3. Lightweight and energy-efficient ML library (T5.3, Section 3.3.3)

This library is expected to be used within FLaaS to provide:

1. novel ways and mechanisms to perform the machine learning training of FL models on device, using energy-efficient methods (e.g., with early stopping of the training when sufficient performance is achieved);
2. new communication protocols for initializing and managing FLaaS tasks, that will allow the reduction of overhead while transmitting the local and global models between end-user (or edge) devices and the server/backend.

This library will allow the FLaaS developer to adjust the processing demand of the learning tasks and their energy footprint. This can be achieved through library's modules that make use of the following methods:

© 2023-2025 TaRDIS Consortium

- Knowledge distillation:
  `studentNN = KD(teacherNN, NNtype, Nhl, accuracy, td)`
  where the inputs and output are described in the related Section 3.3.3 above.
- Pruning:
  `newNN = pruning(baseNN, compr_rate, accuracy, speedup, pr_method, td)`
  as described in the related Section 3.3.3 above.

4.4.4. Decentralised membership and communication (T6.1, Section 3.4.1)

This task will provide 2 services that are useful for the new version / generation of FLaaS:

1. decentralized membership service that will help FLaaS, under its hierarchical version, to maintain information at the intermediate/overlay layer about the end-user or edge devices participating in the FLaaS platform and which of them can execute an FL task at any given moment, as well as which devices can take the elevated roles of super-peers (intermediary nodes in the overlay) given that there is heterogeneity expected in the resources available per node;
2. communication service that will provide point-to-point and point-to-multipoint communication primitives with different guarantees to the FLaaS third-party application developer.

The main function of this API would be the method:

```
create_overlay(functionality, type (optional))
```

that is described in the related Section 3.4.1.1. above.

Input parameters except the specified ones would be:

- `NetId` as defined in Section 4.4.1 above,
- `Conn` as defined in Section 4.4.1 above,
- `Sec_conn`: the connections flagged as secure, output of the `establish.sec.connections` function described in Section 4.4.1 above.

Then, the functions described above for subscribing, publishing, multicasting, etc. can be applied on top of the overlay network.

4.4.5. Decentralised monitoring and reconfiguration (T6.3)

This task will help FLaaS to support FL task execution across heterogeneous and highly dynamic settings. This means FLaaS backend (and super-nodes in the hierarchical version) will use the decentralized telemetry information collected (by the module build in this task) from the different participating end-user and edge devices, to assess health and load conditions, and to make continuous decisions on where to train the FL tasks. FLaaS will also use the solutions designed within this task to perform membership, communication, and replication decisions at the overlay layer (hierarchical FLaaS), as well as the ML training execution on participating FL devices.

```
state_T = current_state(T) # this call (at the time instance T) will give as
                           # output the characteristics of the current system
                           # state
```

The returned `state_T` is an array consisting of (not exhaustive list):

- `NetId` as defined in Section 4.4.1 above,
- `Conn` as defined in Section 4.4.1 above,
- `Sec_conn` as defined in Section 4.4.1 above,
- `B_u` : amount of data per end user u to be processed.

© 2023-2025 TaRDIS Consortium

```
update_overlay(state_(T-1), state_T) # updates the overlay network
```

where the arguments are

- `state_(T-1)`: the state of the previous time instance (T-1) stored in memory and
- `state_(T)`: the current state (T), which is the output of the current_state function.

Output: list of modifications in the overlay network and orchestration of the network. Effective immediately and before T+1. This is closely related to the decentralised membership and communication API (Section 4.4.4) (`create_overlay`). In particular, this API will account for potential failures (as discussed in the Deliverable D4.1, Section 5.1) such as communication failures, imbalanced data distribution, participant dropouts, etc. by making adjustments on the decision variables.

Funded by the European Union

© 2023-2025 TaRDIS Consortium

5 CONCLUSION

This report has documented the ongoing work on the specification of the TaRDIS programming model, and on the specification and development of the TaRDIS toolkit APIs. It has outlined how the APIs made available by each work package (as part of the TaRDIS toolbox) will be leveraged by each project use case. This documentation is an important step towards the project objectives, ensuring the alignment of the various work packages.

The outcomes of this deliverable have been made possible by the close collaboration between the project partners. As the TaRDIS project activity progresses, the TaRDIS programming model and APIs will undergo further consolidation and alignment with the use case and toolbox requirements produced in Deliverable D2.2 (released concurrently with the present Deliverable D3.1). To this end, the TaRDIS project tasks T3.1 (models) and T3.2 (APIs) will continue their activity by maintaining up-to-date documentation of the model and APIs, and fostering collaborative design through workshops. The results of these activities will be documented in the next iterations of this deliverable (D3.3 and D3.5).

Funded by the European Union

© 2023-2025 TaRDIS Consortium