anpa

D7.1: Report on the expected improvements and quantification procedures

Revision: v.1.0

Work package	WP7
Task	T7.1
Due date	30/Sep/2023
Submission date	30/Sep/2023
Deliverable lead	Roland Kuhn (ACT)
Version	1.0
Authors	Roland Kuhn (ACT), Giovanni Granato (GMV), Aravindh Raman (TID), Nicolas Kourtellis (TID), Tiago Teles (EDP), Manuel Silva (EDP)
Reviewers	Ivan Kaštelan (UNS), Sotirios Spantideas (NKUA)
Abstract	This document reports on the findings of implementing the use cases without the TaRDIS toolbox, focusing on identifying and consolidating the areas of improvement that lie within the project scope.
Keywords	use case implementation; programming tools; baseline implementation

www.project-tardis.eu

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

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

Document Revision History

Version	Date	Description of change	List of contributor(s)
V1.0	29/9/2023	first submitted version resulting from concurrently integrated reviews	all authors

Disclaimer

Funded by the European Union

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

Copyright notice

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

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

DEM: Demonstrator, pilot, prototype, plan designs

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

DATA: Data sets, microdata, etc.

DMP: Data management plan

ETHICS: Deliverables related to ethics issues.

SECURITY: Deliverables related to security issues

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

Executive Summary

The following document is deliverable D7.1 of the TaRDIS Project, funded by the European Union's Horizon Europe research and innovation programme under grant agreement number 101093006. It reports on the progress and findings of task T7.1 "Baseline".

We present the baseline implementation work performed by each of the four use case partners, including a description of the respective system architecture and interim results. Special focus lies on identifying the challenges faced in this process, in particular those that lie within the stated project scope of the TaRDIS consortium; we briefly list challenges that we will not improve upon because they are outside the project scope and will thus be tackled according to the state of the art.

The main outcome is presented in section 4, where we consolidate the in-scope challenges into a list of areas of improvement. Each area is presented with an expected improvement from the perspective of a programmer using the TaRDIS toolbox. We also list KPIs that will allow the quantification of the achieved improvement for each of them.

Table of Contents

List of Figures

Funded by the European Union

List of Tables

Funded by the European Union

ABBREVIATIONS

AGV	Automated Guided Vehicle
API	Application Programming Interface
BDS-3	BeiDou 3rd generation navigation satellite system
CDF	Cumulative Distribution Function
DER	Distributed Energy Resources
DL	Deep Learning
DP	Differential Privacy
ERP	Enterprise Resource Planning
FL	Federated Learning
G2G	Galileo 2nd Generation of satellites
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
IPFS	InterPlanetary File System
ISL	Inter-Satellite-Link
JS	JavaScript
LEO	Low Earth Orbit
MES	Manufacturing Execution System
ML	Machine Learning
ODTS	Orbit Determination and Time Synchronization
P2P	Peer-to-Peer
PNT	Position, Navigation and Timing
SGAM	Smart-Grid Architectural Model
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1 Introduction

This document reports on the work of task T7.1 “Baseline” done in the first nine months of the project. As foreseen in the work plan, the implementation of use cases with state of the art tools—before the introduction of the TaRDIS toolbox—has led to deep insights into the programming challenges affecting the field of intelligent swarm systems today.

Section 2 comprises descriptions of the four use cases in terms of architecture, implementation, interim results, and identified challenges. We note that the TID use case has changed slightly in shape relative to the description in the project proposal: the bulk of the effort will be spent on developing a hierarchical federated learning [HFL] toolkit (which will be part of the TaRDIS toolbox), which will then be the basis for a set of smaller use cases implemented for the purpose of validation. An important aspect is that the HFL toolkit itself will be based on the communication and data management tools offered by TaRDIS, meaning that this work also contributes to the inputs for and validation of these parts of the toolbox.

Section 3 enumerate areas that will not be the target of TaRDIS improvement. In the former case these are generic requirements without which the use cases cannot meaningfully be implemented; it is therefore assumed that these properties are always upheld, both in a baseline implementation and the final TaRDIS implementation. In the latter case we note challenges that are either specific to one of the use cases without sufficient potential of generic extension, or problems that already have solutions in the state of the art upon which the consortium in the TaRDIS scope cannot realistically improve with the available resources and expertise.

Section 4 then presents a consolidated view on the identified use case challenges that TaRDIS will improve on, structured according to the work packages in which the respective main body of work will be performed. Each area is briefly motivated and the improvement sketched—the details of the offered tooling will be reported on in further reports especially from WP3. We also give candidate KPIs for quantifying each achieved improvement, noting that the precise definition of these measures will depend on the details of the emerging tools offered by the TaRDIS toolbox and that most of the described measurements have not yet been performed on the baseline implementations that are still under development.

Funded by the European Union

2 Implementation of Use Case Baselines

In this section we describe how the baselines have been implemented: using which tools and techniques, with how much effort, and where the main pain points were.

2.1 EDP

Next, we describe how the baseline has been implemented for the energy use case.

2.1.1 Background Introduction

Energy grids play a crucial role in modern society connecting energy producers with consumers, ensuring a smooth and ideally uninterrupted flow of energy. Maintaining a careful balance between energy generation and consumption is essential for the effective functioning of these grids. In today's context, an increasing number of energy consumers have also become energy producers (prosumers). This shift, driven by motivations such as personal energy ownership and usage or by economic incentives such as selling excess energy to the grid or local neighbours (peers), has led to an increasingly more decentralized energy landscape [14].

This decentralized energy setup requires a shift in how we manage and control it. We're moving away from the traditional centralized approach to a more distributed one. In this new setup, different energy sources and consumers collaborate to share energy, helping to maintain a stable grid. This mechanism of surplus energy from one source matching the energy needs of another, in order to ensure an efficient energy distribution system, is a problem similar to the computer science realm and to how swarm intelligence/collaborative intelligence works. Swarms of energy management devices can ensure grid stability, in an increasingly dynamic grid, where a more centralized system would have difficulties keeping up with the local changes.

2.1.2 Energy Baseline

The starting point of the energy baseline was a concept based on a centralized system depicted in Figure 1.

From the figure, one can observe the exchanges of energy and data. In this early centralized version, citizens can access the data they generate, which is then unidirectionally exchanged with the central system. As for energy transit, it occurs from a central producer—typically a power plant—to the final consumer. The final consumer can also function as a producer, and an orchestrator at the 'Grid' level is responsible for managing and balancing the grid.

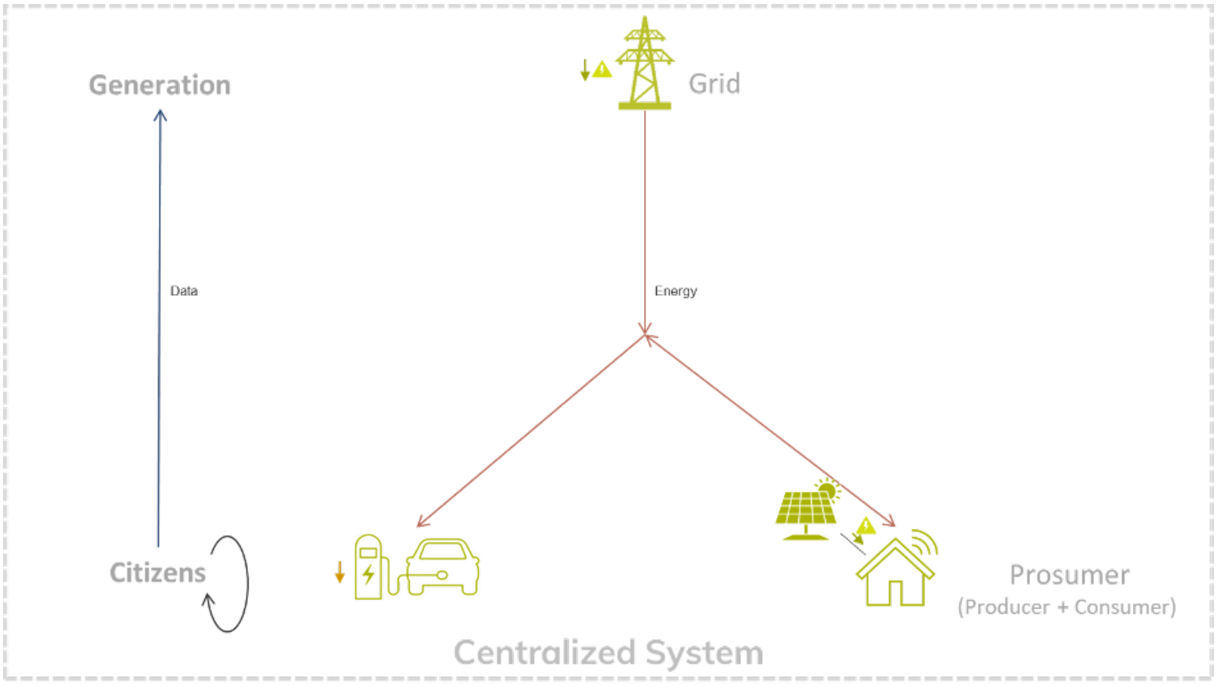

Figure 1: EDP centralized baseline

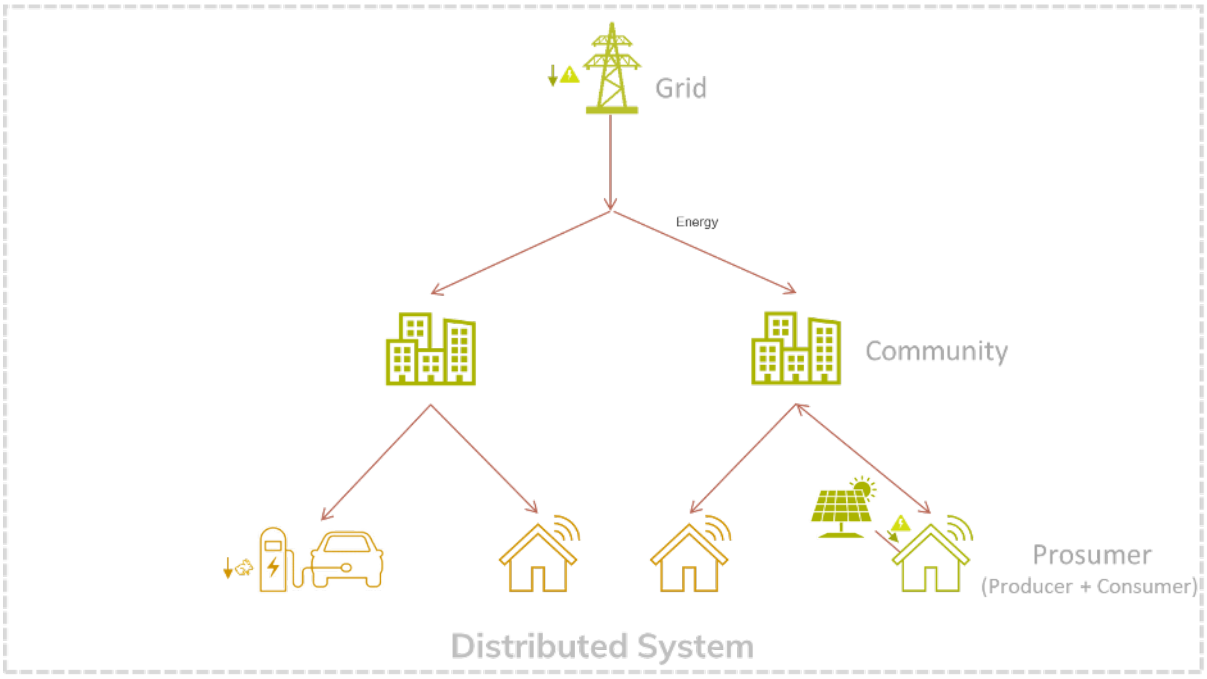

Figure 2: EDP distributed system baseline

The baseline implementation followed the second iteration of the concept, as illustrated in Figure 2. This iteration adopts a distributed approach in contrast to the centralized one described earlier.

In the distributed system, it introduces the concept of an energy community, where the electrical grid remains the same as described previously but is now segmented into three layers. These layers include the national grid layer, followed by the community or neighborhood layer, and finally, the household layer.

Energy flows predominantly in a cascading manner from central generation. However, with the inclusion of a market orchestrator operating between the grid and community layers, imbalances within each community can now be matched and balanced. This enables the utilization of surplus energy in one community, such as Distributed Energy Resources like rooftop PV, to offset the deficits in nearby communities. For instance, a neighboring community may experience higher energy demand, especially when an electric vehicle is charging. The orchestrator can proactively manage these imbalances with one-hour-ahead time slots, thus alleviating constraints on the national grid. The community level orchestrator may also help balance energy production inside of that community, by redirecting an energy surplus from one home to another home requiring that energy.

To implement the second concept in the laboratory, we set up the environment depicted in Figure 3, where data from each component is broadcasted within the community.

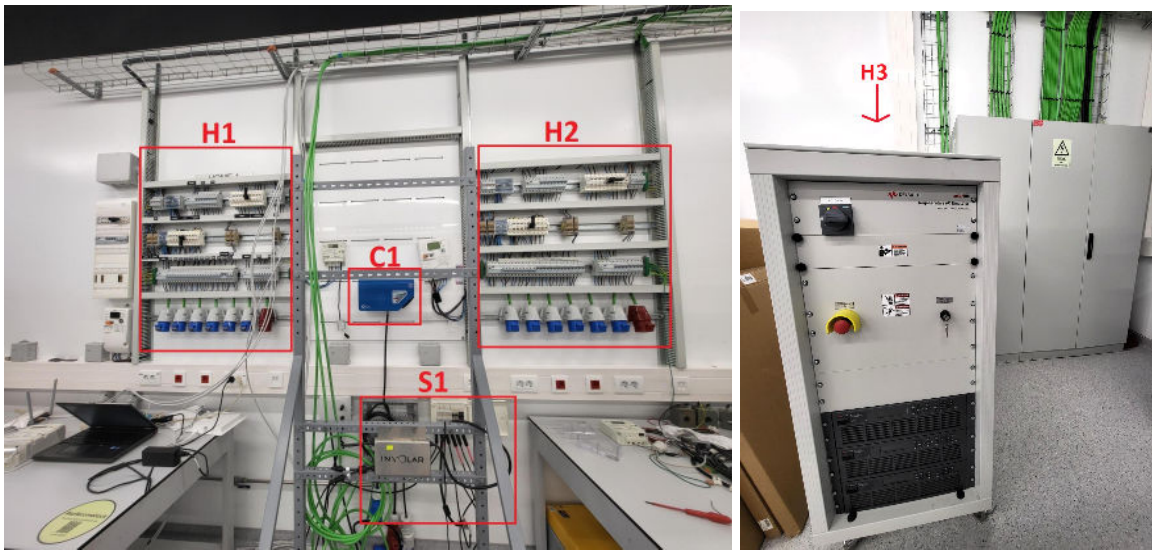

Figure 3: EDP lab setup for the distributed system baseline

This setup consists of three houses, one EV charger, and a PV array. Specifically, we have two houses equipped with typical electrical switchboards (H1 and H2), which serve as real-world representations of the orange ones depicted above. Additionally, there is a unidirectional residential wall box for EV charging (named C1) and a dynamically controllable House H3. The solar array is connected to H3, and it provides real-time data from rooftop PVs, allowing for dynamic changes in energy generation.

Houses H1 and H2 are considered part of different communities, with each house connected to the national grid. The grid emulator is capable of communicating via API with the market operator.

This market operation mode uses the more traditional client-server architecture, with a particular case of the Community level being both a client and a server at the same time. It is a client to the grid, but a server to each home.

This lab environment serves the purpose of conducting initial baseline validation, this initial validation does not use any TaRDIS component.

The final concept, depicted in Figure 4, represents a distributed and more dynamic approach where each player can communicate with every peer, this solution significantly benefits from the outcomes of TaRDIS.

In this scenario, the orchestrator will possess the ability to aggregate information from the three layers, transitioning from centralized control to a distributed control model, similar to the operation of a Virtual Power Plant. Advanced cryptography usage, such as group signatures, maintains the privacy and necessary anonymity of users, while guaranteeing no foul play or market disruptions, leaving the energy market holder with the possibility to uncover and diagnose messages from misbehaving nodes in the network. This shift, combined with more accurate forecasts for energy generation and consumption, will enable new business models.

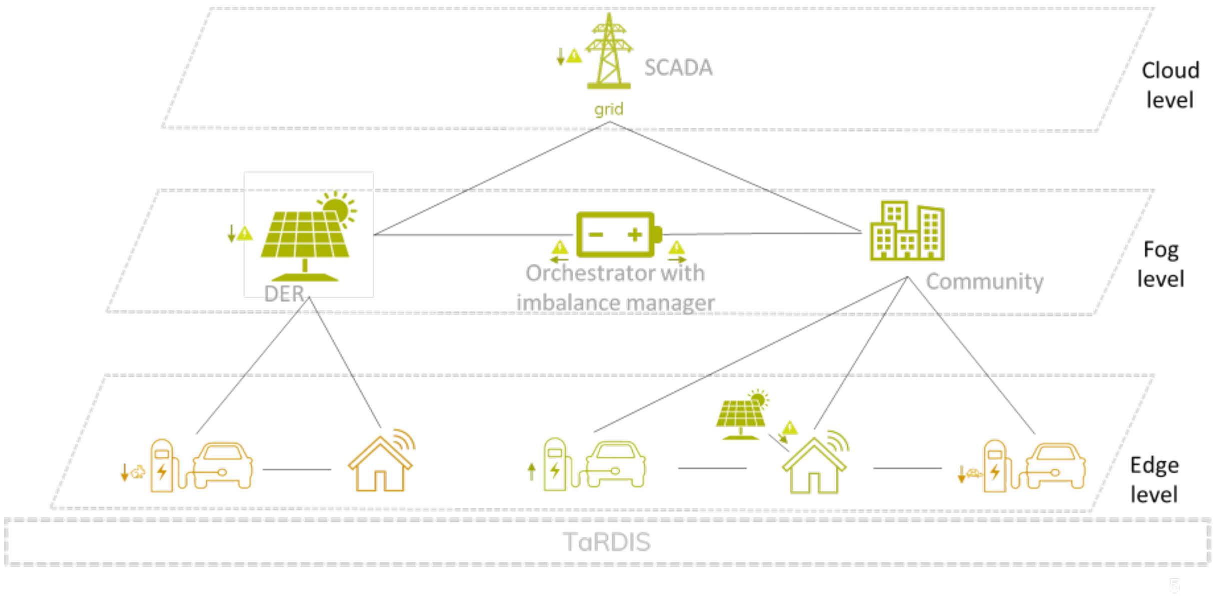

Figure 4: EDP final concept

To summarize this section, considering Figure 4, the introduction of TaRDIS is anticipated to bring several benefits. TaRDIS outcomes will include consumption and generation forecast as a way of short-term grid balance planning, data layer in the communication model within the community to coordinate offers and needs and also an abstract way of programming all of this.

The benefits for the Energy sector include enhanced capabilities for establishing new energy communities, improved grid operations, and increased resilience due to the finer granularity in measurement and control. Another relevant outcome is the potential maximization of locally produced energy, which can have a positive environmental impact given its renewable nature and reduced distribution losses. Further improvements/benefits are being discussed and aligned with the researchers from academia and IT partners.

To assess these metrics, we employ the use case KPIs as the primary measurement indicators. In the environmental domain, it becomes feasible to quantify the CO2eq

Funded by the European Union

emissions saved when compared to a typical fossil fuel alternative. Lastly, for the set up of new communities, this verification will be carried out through the utilization of a TaRDIS developed application designed to execute this task.

2.1.3 Remaining difficulties

As of today, cf. from Figure 2, the energy management is done manually by an aggregator which is part of the Energy Community and the grid operator, both working for grid stabilization—grid energy stabilization guarantees that all consumers are supplied. Communication between grid operator and aggregator is not automatic.

In TaRDIS one still needs to establish the procedures first, for matching the request and offer of energy before the transaction of energy itself and then the strategies for handling real time faults either for consumers' or producers' side.

Technically, there are some challenges that still need to be overcome to get this system into production. Most of these are covered by a comprehensive TaRDIS system.

The identified challenges are:

1. Interfacing with electronic simulation hardware programmatically. More coordination between EDP departments in order to develop a useful API for electronic simulation hardware is still ongoing.
2. Advanced cryptography usage was a complex issue to solve for the EDP use case, as some more obscure, but extremely useful cryptography schemes for anonymous, private, yet secure peer to peer systems are very underdeveloped (such as group signature schemes).
3. Comprehensive network hole punching. To develop a system where the nodes truly communicate peer to peer, a network hole punching solution has to be developed within the TaRDIS toolkit. A network condition agnostic hole punching solution would make real world usage of swarm systems, through the public internet, much more reliable and faster to execute.
4. The implementation and testing of distributed network overlays are very time consuming, error prone, require extremely specialized labour and are a commonly duplicate effort between different projects. A toolkit where it is possible to specify the properties that are required for a certain overlay, and one is transparently chosen and deployed would make this step much faster to iterate and develop on.
5. The verification of correctness for distributed systems is also a particularly difficult challenge. Although the correctness may be approximated empirically, by running tests, a mathematical guarantee of correctness (lack of deadlocks, lack of infinite message loops, provably correct encryption, etc) would ease the deployment of swarm systems into the real world.

The first two are specific to the EDP use case and lie outside of the scope of TaRDIS. Network hole punching is an issue for peer-to-peer systems operating over the Internet, where participants are not all in the same local network. Solutions to the unfortunately common violations of the Internet's original promise of *end-to-end connectivity* are being developed by many organisations, including Protocol Labs to whom we have an ongoing relationship; we plan on using their libp2p implementation since providing a proprietary solution is out of scope for TaRDIS.

2.2 TID

2.2.1 Background Introduction

Decentralised and Federated Learning (FL) [1], coupled with Differential Privacy (DP) [2], Trusted Execution Environments (TEE) [3], or other methods, enables data owners to securely train a Machine Learning (ML) model among them, by only sharing private models trained locally on their data. In fact, FL has been adopted by major tech corporations (e.g., Google, Apple, Meta, etc.) due to properties such as privacy-preserving (PP) data modeling, scalability and performance.

Typical FL use cases currently employed by industry focus on building an ML model for a specific ML task by applying FL across user devices, i.e., cross-device, on user local data (e.g., GBoard). These deployments are based on proprietary code, embedded in ecosystems, and typically designed on a per application (app) basis. This clearly limits the usage of FL to a few big players able to sustain cost and risk of developing and using FL.

To change this status quo, recent start-up or open-source efforts aim to provide FL tools for B2B customers or end-users, where FL is performed across company servers, i.e., cross-silo, on cloud-collected data (e.g., between health providers, or financial institutions, etc.). Despite these early efforts, there are several novel scenarios where FL can be applied that remain unexplored, such as the cross-app mode: a combination of cross-silo and cross-device for an ML task that apps share. The below motivating examples share the same scenario: two or more apps collect partially similar data (they may be independently collected and annotated with labels by each app and its user) and want to collaborate and solve the same ML problem with better performance (e.g., accuracy):

- Health: A fitness and a nutrition app want to detect when their user is at risk of diabetes and alert them. Each app may have access to complementary data, or exactly the same data (e.g., blood pressure, type of exercises performed, calories consumed, etc.), and their combination may allow the apps to find out more accurately if their user / data owner is closer to such a health risk.
- Entertainment: Video apps (e.g., YouTube, Netflix, etc.) want to train a better recommender system for their customers. Same scenario for audio apps (e.g., Spotify, Pandora, etc.).
- Image Analysis: Apps analyzing images for object classification (e.g., Google Photos, Android Gallery, etc.) want to build a model for better detecting objects, contexts, etc. The labels (i.e., objects present on each image) are assumed to be provided locally by the user / FL device owner, and can be used for training of the FL model.
- Ad delivery: Ad providers want to build a model to better predict when the user is more likely to click on ads, delivered through push notifications.

In these, and many other scenarios, if the apps could locally share their data or ML models in a PP fashion, they could train a unique FL model across their users' devices that helps all participating apps, while respecting data owner's privacy. Under the cross-app mode, even multi-task ML problems can be collaboratively solved between apps. Unfortunately, crucial system research challenges remain to be solved to materialize such cross-device cross-app scenarios in the mobile context, and even democratizing FL to small and medium companies in as-a-service fashion.

2.2.2 Baseline Architecture & Implementation

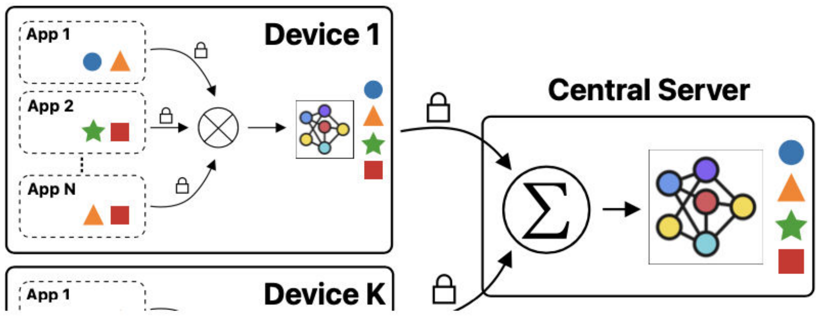

Figure 5: Implementation Overview of FLaaS.

In the baseline implementation, we build FLaaS, the first practical federated learning framework for mobile environments that enables cross-device and cross-app (i.e., on-device cross-silo) FL, and as-a-service. In Figure 5, we show in FLaaS how K devices with N apps installed across the system, are securely sharing their heterogeneous samples or locally trained models with the local service. These are aggregated locally (denoted by ⊗) in order to produce a local model. The model parameters from all K devices are securely aggregated by applying FedAvg (denoted by Σ) following typical FL protocol.

FLaaS has been designed to address the following FL system challenges:

- Existing FL methods do not provide an easy to use way for app developers based on simple APIs and tools as ML-as-a-Service counterparts (e.g., Amazon Web Services [4], Google Cloud [5] or Microsoft Azure [6]). To address this challenge, for the baseline implementation, we use a centralized, cloud component to automatically orchestrate and load balance the FL server, with an easy user interface (UI) for app developers, and scalability for multiple parallel FL tasks. Our design is end-to-end and includes several system optimizations required to deploy it in a scalable, robust as-a-service fashion.
- There is a lack of libraries to support third-party app developers to train and federate models in the background. Therefore, developers either train models while users are active on their apps, directly impacting user experience, or rely on the OS scheduler, which has full control in orchestrating such heavy compute tasks in the background, but with well-known restrictions and limitations. To address this challenge, our design includes a client-side middleware (Sec. 2.2.3) and library enabling third-party mobile apps to work with such OS constraints, and train FL models on-device.
- There is no support for cross-app local sharing of data or models for collaborative ML training: it requires secure and PP mechanisms for storage, communication, and permissions management across apps. To address this challenge, within the library, we offer two types of cross-app FL modeling: joint (shared) samples and joint (shared) models. This library, based on TensorFlow Lite, can be integrated in existing

apps, and used by calling a set of secure and PP communication and storage primitives and APIs. In particular, a secure communication channel is established between apps embedding the library and the service supporting different types of data, i.e., numerical, text and binary. Also, data and model storage on devices are secured using per app and service spaces with permissions specified by developers and enforced using an on-device data management engine.

For the sake of simplicity, we will call our service FLaaS (FL-as-a-Service) that allows app developers to perform cross-device, cross-app federated learning. We implement the FLaaS for Android-based mobile devices (i.e., client-side elements), while leveraging a popular cloud-based platform for the centralized component. Figure 6 provides the FLaaS architecture overview, along with the list of steps required to perform an FLaaS training process. As illustrated, there are four core components of FLaaS: 1) App Developer Interface 2) FLaaS Server 3) Notification Service and 4) Client Devices.

In FLaaS, we assume a set of client devices participating in FLaaS device infrastructure, periodically reporting their status through the backend's REST API (1) to the DB for logging (2). An app developer uses the Admin Interface to create a new FL project (3), whose configuration is stored in the DB (4). The Device Scheduler detects the new FL project (5), queries the device statuses from the DB (5), and sends an FL training request using its Notification Service provider (6a) to external services such as Firebase Cloud Messaging (FCM) (6b). Each device's Local module receives the FL training request (7) and requests from the participating third-party apps to receive either their local samples for training, or to train their own models (8a, b and c). It then conducts the necessary on-device model training (if it received samples) or model aggregation and averaging depending on the training mode. When a substantial number of reported models is received by the Server (1), the Device Scheduler instructs the Model Aggregator to accumulate the received models (9a and 9b), marks the FL round as complete, and continues with the next FL round until the project is complete (i.e., stopping criteria are reached).

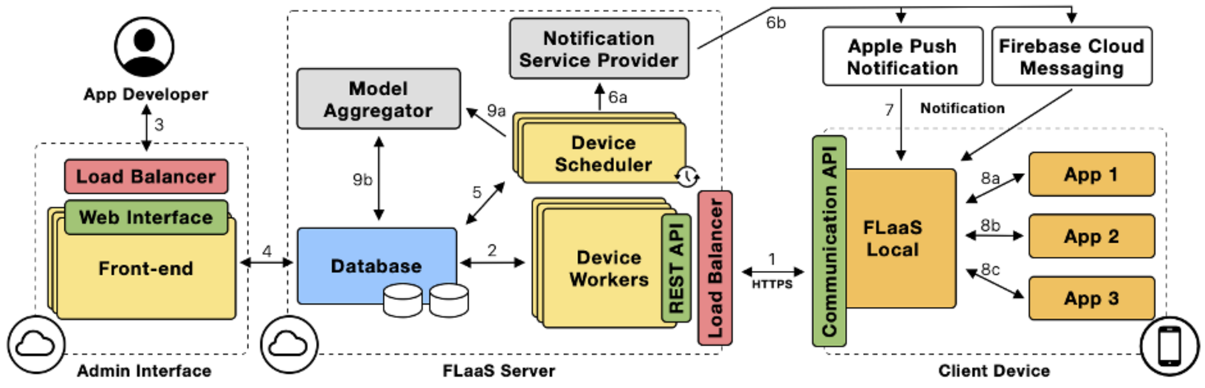

Figure 6: Overview of FLaaS architecture with task execution steps.

Next, we explain each of these four core components:

1. App Developer Interface. The first function that a service such as FLaaS should offer is an admin UI for app developers, that allows them to bootstrap, configure, monitor, and terminate an FL project executed within the system. Developers can use this UI for registering their apps with the system, produce the authentication tokens that will be used in their app and configure the data and model access policies to be enforced by FLaaS at device level. Thus, they can create new FL projects to be executed from FLaaS-registered devices, make a pre-selection of participating apps and devices, and initiate the FL training process.

Besides the typical FL parameters available in FL system (e.g., ML model architecture, epochs, number of FL rounds, FL model loss, learning rate, partial participation ratio, etc.), our system provides additional configuration for cross-app training that each app developer can configure separately, such as: training mode (joint samples or models), policy management (app data access, expiry date) locally per client, etc.

When the developer defines a new project using the UI (step 3, Fig. 2), the project is pushed to the FLaaS Server (step 4) for (i) storing its configuration, (ii) executing the project in the available client devices, (iii) monitoring the project's health and progress, (iv) terminating the project when stopping criteria are reached.

2. FLaaS Server. This is a cloud-hosted service in charge of executing the FL project previously created at the App Developer Interface, while coordinating the Client Devices. It consists of typical FL system modules: 1) Database, to store project-related metadata and models received from devices; 2) Device Schedulers, to monitor and manage project health and status through time and devices; 3) Model Aggregator, to apply aggregation functions as requested by app developers. 4) Device Workers, to interact with client devices for FL model training assigning and responses received via REST endpoints, all exposed to the client devices using token-based authentication, e.g. to report device availability and status updates (e.g., battery level, charging state, etc.), to retrieve the FL model parameters, to join the FL round, to submit the FL model parameters, to submit performance results, etc. It also includes a load balancer module to evenly distribute load across workers. 5) Notification Service, for sending FL training requests from the Server to participating Client Devices in a secure, asynchronous, and one-way communication request (details below).

3. Notification Service. The Notification Service (NS) is used for secure, asynchronous, and one-way communication from the Server to Client Devices. While it can generally depend on various types of technologies (e.g., message brokers, TCP connections, etc.), in our context, it takes the form of a Push NS. These are cloud-based, highly efficient services for Android and iOS to propagate information to a previously registered mobile app, either in the form of a visual popup, but also invisible (silent) notifications that deliver JSON structured data to an app. They use secure TCP communication to directly push data to a device's registered app through dedicated cloud services provided by Google (FCM) and Apple (APNs). Silent notifications (i.e, that do not appear to the user but wake up the app in the background) are usually received with an unspecified delay from the app. We implement the backend service within Heroku Cloud Application platform for hosting the server modules, PostgreSQL and Amazon S3 data store for storing, Django REST framework for Device Workers, and Pushwoosh for the Notification Service.

4. Client Devices. The most important piece of FLaaS is the set of client devices in charge of computing the ML training tasks orchestrated by the Server for the execution of an FL project. On such devices, there are two main modules: Local and Library. These modules facilitate novel on-device functionalities: App authentication, inter-app communication, access policy management, ML model training, data storing, and status reporting.

4.1 FLaaS Local. This is a trusted, standalone service that needs to be installed on the device and provides the core FL functionality to the device in accordance with requests sent by the Server. We do not make any assumption on how this service lands on-device: it can be pre-installed on the device by the OS provider or manufacturer. Alternatively, an interested third-party app developer can recruit a device owner to install its app with proper user incentives (e.g., promise of better app experience due to model personalized on user data).

Local provides authentication and secure communication of the device to the Server, and periodically reports the device status (akin to a heartbeat), with different details such as battery level, charging state, or connectivity state, to the Server for consideration. The set of statuses from all participating client devices is in fact the one collected and analyzed by the Server to decide which devices to invite for upcoming FL training tasks. Local is also the module that receives messages from the Notification Service with configurations for pending FL project tasks related to specific third-party apps. These task configurations are then communicated to the appropriate apps for execution, within their Library (see next).

Table 1: Core software components used during FLaaS client development.

Supported Mobile Operating System	Android (≥0.8), iOS does not provide (yet) in-app communication.
FLaaS Library	LoC: 1.5k TensorFlow Lite v2.6.0 WorkManager v2.7.0
FLaaS Local	Standalone android application (Java) LoC: 2.4k Retrofit v2.9.0, Pushwoosh Android SDK v6.3.5
Inter-App Communication	Android Broadcast Receiver
App Workers	WorkManager API

4.2 FLaaS Library. This is a mobile app library integrated within each FLaaS-enabled third-party app willing to participate in FL training processes. This library has a set of simple APIs that are utilized by the developer in the app code. More particularly, the API consists of calls to register the app's secure token (created through the App Developer Interface) and to provide controlled (secured through policies) access to data the app is willing to share. Under the hood, it implements a set of functions that allows the app to securely communicate with Local to receive ML modeling configurations based on FL project of relevance to the app, execute the appropriate on-device ML training, and share data or model parameters with Local. Table 1 lists various software components used to build the FLaaS client devices.

While cross-app FL modelling is enabled, FLaaS has three modes of operation:

Single-app FL modeling: In this case, each app in the device trains a model on its own data only. The starting configuration for this training is received by the Library of the specific app, and includes the base model to be trained, access policy requirements on the local data to be used, and model hyperparameters. If the developer defined DP-noise to be injected, this is done at this ML training stage. After local training is done, each app sends to Local its model built. When Local has all such models from unique apps, it compresses and transmits them to the FLaaS backend server. The backend performs FedAvg across all reported local models of apps and builds one global weighted average model per app, which it then communicates back to the participating devices. Finally, Local distributes this global model to each app for the next round.

Joint Samples (JS) mode: The apps share training samples with Local, appropriately packaged to a format pre-agreed by the developers. DP-noise is added by each app, in a pre-training stage. Local checks the received samples from each app for formatting, and then merges them before performing ML model training on all. If these third-party apps require it, Local may also inject further DP-noise during ML training. Then, Local communicates with the Server the resulting model as in single-app mode.

Joint Models (JM) mode: The apps conduct ML model training individually using their local samples, as in Single-app mode, with the defined DP noise. Then, each app shares only its model with Local. Local applies FedAvg to all received models and produces a joint model. It finally communicates to the Server the joint model across apps of the devices. Global performs FedAvg across all collected joint models and builds a global weighted averaged model, for each collaborating group of apps. Then, it communicates each such global model back to participating devices. Finally, each Local distributes the global model to each app of the collaborating group.

2.2.3 Baseline Interim Results

We evaluate the FLaaS feasibility, overhead and ML performance via in-lab experiments from an image classification-based use case (i.e., an image represents a specific object such as car, truck, airplane, etc., and the ML task is to predict what that object is). The in-lab tests allow us to evaluate the cost of the system under controlled settings: we observe devices spend a limited amount of time (i.e., few minutes) and resources (i.e., energy and CPU) in training of an ML model during an FL round, and communication with FLaaS. Further, we test FLaaS model performance on a cross-app use case: *image classification* for image analysis. The cross-app scenario materializes assuming three toy mobile applications that need a model to perform the same ML task (image classification), and decide to collaborate and train a better ML model.
The evaluation with in-lab experiments aims to ensure the feasibility of FLaaS being deployable on Android devices and to measure the cost of FLaaS operations from the point of view of latency, CPU, and energy consumption.

Example Application. We create three simple apps (named Red, Green, and Blue, or RGB), with 101 lines of Java code each. Each app includes FLaaS Library v0.1.0 for supporting system functions and to demonstrate the simplicity and efficiency of a third-party app executing FLaaS projects. In the next experiments, apps were preloaded with CIFAR-10 data [7] and executed image classification tasks.

Performance Metrics. We assess the cost of on-device FLaaS functions by measuring compute cost via CPU utilization (in %), time delay (in minutes), and energy consumption (in mWh) taken to compute said ML tasks. We also measure performance of the ML model trained using test accuracy as a standard ML utility metric.

System Costs: We begin to assess on-device cost for 3 fundamental FLaaS functions executed during an FL round, by measuring time required to execute them on user devices:

- Join FL round: Communicate with Server and download global FL model parameters.
- Model Training: Load local samples into on-device ML engine and conduct the training.
- Model Aggregation: Aggregate local models and apply FedAvg (in JM only; typically, very small).
- Reporting: Communicate with Server to report updated local model and other performance metrics.

To measure the system cost of on-device ML training, we perform several experiments with three popular Google Pixel devices (specs in Table 2) as test devices in a controlled setting: the devices are forced to perform the ML task on Independent and identically distributed data, from beginning (received notification of new ML task) until the end (delivery of ML model to Server) without any breaks. All devices were updated to Android 12 and with a deactivated OS-related automated process that would influence measurements (ie., automatic updates, adaptive battery and brightness).

Table 2: Specifications of devices used in the in-lab experiments.

Model	*Device Specification*
Pixel 3a (P3a)	Snapdragon 670 (8-core, 10nm), 4GB RAM
Pixel 4 (P4)	Snapdragon 855 (8-core, 7nm), 4GB RAM
Pixel 5 (P5)	Snapdragon 765G 5G (8-core, 7nm), 8GB RAM

During evaluation, each device was connected to a stable 5GHz (IEEE 802.11ac) Wi-Fi network, while CPU utilization was collected using the Android Device Bridge (ADB) tool, and energy consumption (in mWh) using an energy measurement infrastructure at TID [8]. All RGB apps are whitelisted (i.e., set to Unrestricted Battery usage) aiming to measure the most optimal scenario where all apps are responsive to ML training requests, without further scheduling delays introduced by Android's Work Manager.

Table 3: Performance of FLaaS training modes on devices in lab experiments.

	Duration (sec)		**CPU usage (%)**		**Energy (mWh)**	
	JS	**JM**	**JS**	**JM**	**JS**	**JM**
	Mean (SD)	**Mean (SD)**	**Mean (SD)**	**Mean (SD)**	**Mean (SD)**	**Mean (SD)**
P3a	167.3 (13.1)	85.3 (1.5)	17.9 (2.1)	29.7 (0.6)	56.8 (8.6)	37.8 (0.5)
P4	117.1 (1.1)	62.1 (0.8)	15.4 (0.1)	27.8 (0.4)	41.3 (0.9)	29.7 (0.5)
P5	115.0 (0.4)	62.8 (0.9)	16.1 (0.1)	28.8 (0.4)	42.7 (1.0)	32.4 (0.7)

Table 3 shows the average time taken for a device to perform the ML training task, for the three test devices and two cross-app modes (JS vs. JM) with a total of 450 samples. As expected, the older device (Pixel 3a) takes the longest to finish, regardless of modeling mode. Also, confirming results from the user study, JS is slower than JM since apps must first transfer data to Local, and then Local has to train the model on them. On the other hand, JM builds 3 individual models, even in parallel if the OS's Work Manager allows it. Interestingly, this parallelized execution in JM is reflected in CPU usage, which is, on average, 74.7% higher on JM than JS, across devices. Finally, the reduced execution time, even at higher CPU usage, leads to overall reduction in consumed energy from JM: 41% lower than JS.

ML model performance: We measure the ML performance of the trained model for the same ML task as before (i.e., image classification), using two FLaaS settings: 1) training a model in a single-app modeling setup, with 250 samples; 2) training a model in a joint-samples modeling setup, using 250+250=500 samples from 2 of the local mobile apps. From these preliminary experiments, we find that sharing data between apps in the joint samples cross-app scenario helps the overall ML utility by 10%, as measured with test accuracy in Figure 7. Also, in further experiments that compared performance of joint samples vs. joint models mode of training, we find that the shared data samples (JS) among 3 apps can achieve up to 48% higher test accuracy across different data distributions when compared with the shared models (JM) approach among the 3 apps. However, these results need to be taken in association to the previous results on cost on devices for executing the various FLaaS ML tasks, with respect to CPU usage, time execution and energy consumption.

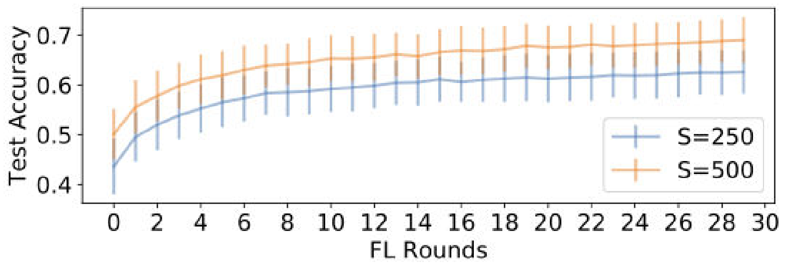

Figure 7: Test accuracy of FLaaS ML model on 2 settings.

2.2.4 Challenges

Privacy and Security of AI models: Privacy-preserving techniques are crucial in FLaaS. Ensuring that sensitive data remains confidential during the training process is quite challenging. Model inversion attacks and membership, attribute, and data inference attacks are privacy threats in FL, where attackers try to extract information about individual data points from the model's updates. Therefore, a challenge to be solved within TaRDIS will be to provide the necessary secured authentication and communication between the back-end and training FL devices, as well as the appropriate aggregation methods that are necessary to protect model updates during transmission.

Furthermore, FLaaS at this point does not provide yet privacy-by-design to the models built. Thus, another challenge will be to find an efficient way to inject differentially-private noise to the models built, in order to protect them from adversarial attacks.

Trust on Backend Server: While FLaaS aims to distribute trust, the central server or aggregator still plays a crucial role. Ensuring the trustworthiness of this central entity is important, and a key challenge to be resolved in the future.

Hierarchy of FLaaS architecture: FLaaS is envisioned to be scalable to thousands or millions of devices. However, this vision raises the question of effective scalability of the central components of the current architecture. Therefore, making use of naturally occurring hierarchies of trust already existing between FLaaS clients and server (e.g., intermediate nodes such as routers, antennas, switches, edge devices, home personal assistants, etc.) can help improve scalability of the FL process execution onto multiple devices, without penalizing performance of the FL model or time it takes to construct it. In fact, using such intermediate nodes could offer opportunities for modifying the trust model of FL clients, i.e., by relaxing the need for trusting only the FL server, and instead being able to trust an intermediary node to perform FL aggregation, privacy noise injection, etc. Therefore, a challenge to be solved within TaRDIS can be to understand how to construct, coordinate, and manage an overlay of intermediary nodes that can help with the FL process, in a way seamless and transparent to the developer.

Secured communication channel between the endpoints: Communication between FLaaS endpoints (FL server, clients, notification service, other intermediaries, etc.) needs to be performed in a secure fashion, regardless of churn of endpoints, scalability of FL server, etc. Therefore, a challenge to be solved within TaRDIS can be to explore existing state-of-the-art means to enable such secure, correct-by-design communication, in a seamless and transparent way to the developer.

Heterogeneity of FLaaS clients: Clients in an FL setting can have different data distributions, hardware capabilities, and network conditions. Coordinating these heterogeneous clients effectively, and in a correct-by-design fashion can be another major challenge to tackle within TaRDIS. This is especially true when these clients have different roles to perform, such as the one of a regular FL client who just trains its personalized FL model, vs. a client who participates in the hierarchical overlay envisioned within TaRDIS, and is responsible for providing better privacy to the overall FL model built, by aggregating intermediate results from FL clients, protecting them with differentially private noise, and reporting them to the next hierarchical layer (FL server or not) for further aggregation.

2.3 GMV

2.3.1 Background

Orbit Determination and Time Synchronization (ODTS) is the process of estimating satellite position, velocity and clock parameters. ODTS is typically performed on-ground for the GNSS constellation of satellites. It is typically performed by means of either batch least squares technique or Kalman Filter in a centralized way [9]

The GMV use case consists of performing distributed autonomous, on-board, and real-time orbit determination and time synchronization for a large constellation of satellites in LEO.

- **Distributed** means that the processing is not performed on a single satellite but every satellite computes its ODTS solution and shares it with its surrounding visible satellites.
- **Autonomous** means that it is performed with limited ground stations support.
- **On-board** means that processing is performed on-board the satellites and not on ground.
- **Realtime** means that the navigation filter for the ODTS processing can only rely on past and present measurements to compute navigation outputs.

The most promising technology enabling the achievement of this use case is represented by the Inter-Satellite-Link (ISL) communication and ranging capability. This is crucial to share data between the satellites (swarm nodes).

Examples of on-board autonomous ODTS using ISL are the GPS AutoNav system which is present on the block II-R GPS satellites [10] and the third generation BeiDou navigation satellite system (BDS-3) [11]. These autonomous navigation concepts are not yet fully operational. Galileo 2nd Generation of satellites (G2G) is also planned to be equipped with ISL systems.

The initial approach towards the development of distributed ODTS is the centralized ODTS first.

2.3.2 Baseline Architecture & Implementation

The centralized EKF (Extended Kalman Filter) was implemented as a baseline using Matlab/Simulink. The high level software architecture is described in the following figure.

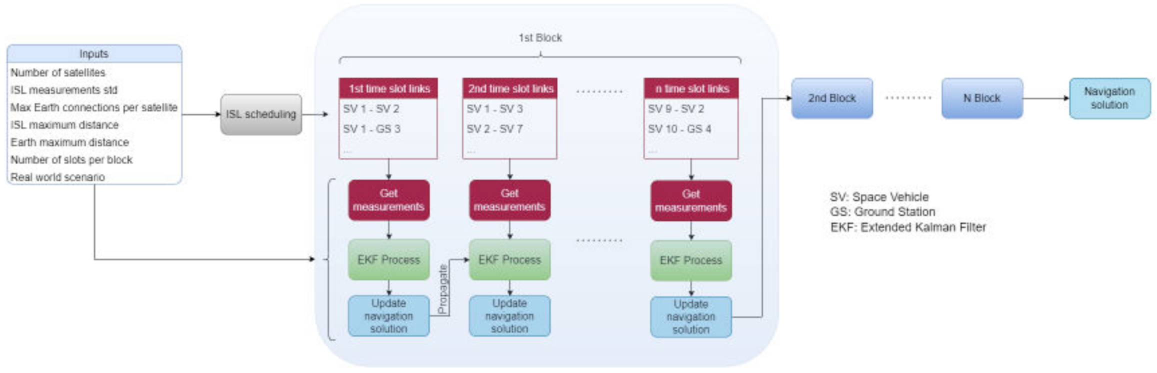

Figure 8: Baseline implementation of centralized EKF for ODTS

As seen from the illustration, a TDMA (Time Division Multiple Access) approach is used for scheduling the observations: each satellite has a specific time window/slot to establish connections with other satellites/ground stations.

The software allows the user to set different types of input parameters such as:

- Measurements noise standard deviations (both range and range-rate): these have a significant impact on the navigation solution accuracy.
- Number of constellation satellites to be considered for the ODTS estimation: for testing purposes it is possible to perform ODTS for a restricted number of satellites
- Maximum number of Earth connections per satellite.
- Maximum ISL distance for establishing connection.
- Number of ISL connections per satellite: this depends on the number of antennas the satellites are equipped with. This parameter is thought to have a significant impact on the navigation solution accuracy.
- Number of time slots per block.

Main assumptions considered during first analyses:

- Each satellite can only connect to one other satellite and one ground station simultaneously (assuming one antenna for ISL ranging/communication and one antenna for ground station connection).
- A ground station can connect to multiple satellites at a time.
- During one time slot both satellites compute relative range measurements (dual one-way measurements). When measurements are combined for processing it is assumed that they are related to the exact same time instant.
- Range and range-rate measurements are affected by white Gaussian noise.

2.3.3 Baseline Interim results

An example constellation of satellites has been defined to be used in the first set of analyses. The selection was based on several criteria, thought to be representative of the target performance for the final configuration. In particular, a **sun-synchronous orbit** (SSO) with an altitude of **1200 km** was chosen, and the number of satellites and orbital planes was adjusted to ensure that every point on the surface had **at least 4 satellites in view at all times** (assuming a **minimum elevation of 10º**). This trade-off, aiming to look for the smallest constellation possible that fulfilled these conditions, resulted in a set of **170 satellites**, evenly spread into **10 orbital planes** in a **Walker-delta configuration.**

The main reference used for the first centralized EKF algorithm implementation is [12].

Among the several parameters studied for each ODTS simulation, there are:

- Simulation execution time: for tuning the ODTS algorithm parameters it is required to run several simulations for testing purposes. The time required for running a simulation plays an important role for the overall development time.
- Computational effort: this is fundamental in view of the space HW development. It needs to be feasible.
- Time to reach convergence: time needed by the algorithm to converge to a stable solution (with error threshold to be established).
- Error after algorithm convergence: this allows to judge the performance of the algorithm used.

2.3.4 Challenges

The centralized navigation involves estimating the satellites states all at once. This implies dealing with matrices whose dimensions are proportional to the number of satellites. Hence, the computational load increases significantly as the number of satellites of a constellation increases. Moving from a centralized approach to a distributed approach would allow splitting the problem in several local smaller problems handled by each satellite. As an example, assuming that each satellite state vector is characterized by 7 components (3 coordinates for position, 3 for velocity and 1 for time), the covariance matrix of the global state vector (all satellites together) has a dimension of (7*N) x (7*N) with N being the number of satellites whose states are estimated. On the other hand, if each satellite performs its own orbit determination (state vector estimation) it only has to handle a 7 x 7 matrix which would be much easier for a space hardware. Therefore, the idea behind the distributed approach is that each satellite shares the results of its ODTS with its surrounding satellites, which in turn do the same with others.

Another aspect is the implementation of the propagation step in the Kalman Filter. The dynamical model is a non linear differential equation which is numerically integrated. The onboard dynamical model used for propagating the spacecraft position and velocity is quite simple in order to be as light as possible computationally speaking. However, being simple, it does not allow to properly model all the perturbations acting on a satellite, especially in the case of Low Earth Orbits. A machine learning model could be trained to substitute the integration step required for the on-board orbit propagation. This would not only allow faster processing but also lead to more accurate results because a more complex dynamical model could be adopted instead of a simple one. Evidence of the promising effectiveness of ML in orbit propagation is reported in [13]. During TaRDIS project activity this aspect is going to be researched. Moreover, in the decentralized ODTS version, provided that ML models are going to be adopted, TaRDIS is expected to help with the implementation of federated learning techniques to facilitate satellites on-board ML models additional training in a collaborative and more efficient way.

The scheduling of observations is currently performed by a function (ISL scheduling). Many combinations of inter-satellite links and ground stations links are possible, each corresponding to a specific geometry. An optimization of this function is needed to obtain the most accurate navigation solution according to the measurements scheduling. An ML model could be trained to select the best combination of satellite-satellite / satellite-ground station connections, by understanding the relationship between the satellites links and the navigation accuracy given a specific ODTS algorithm.

The noise affecting measurements is assumed to be white Gaussian. However, we know that this is not the case in reality. In order to have a better representation of the specific problem a different algorithm could replace the Kalman Filter. This aspect shall be further investigated.

2.4 ACT

2.4.1 Background

The smart factory use case is being implemented by an Actyx customer in collaboration with Actyx software engineers. The baseline scope is to automate the intralogistics of production lines for the assembly of machining centers: the customer is a company selling machines or whole production cells/lines to other factories. A machining center is a bulky and heavy piece of machinery weighing several tons and comprising high-precision mechanics and corresponding control electronics. It starts out as an empty steel frame at the first production step, advancing every night by a few meters to the next workstation, until after about a dozen days of work all pieces are installed, connected, and tested. The function of intralogistics is to move all parts and materials between workstations and warehouses as required; this includes the components installed in the machining centers under construction, the tools needed for doing so, as well as moving the partially completed machining centers at night.

2.4.2 Baseline Architecture & Implementation

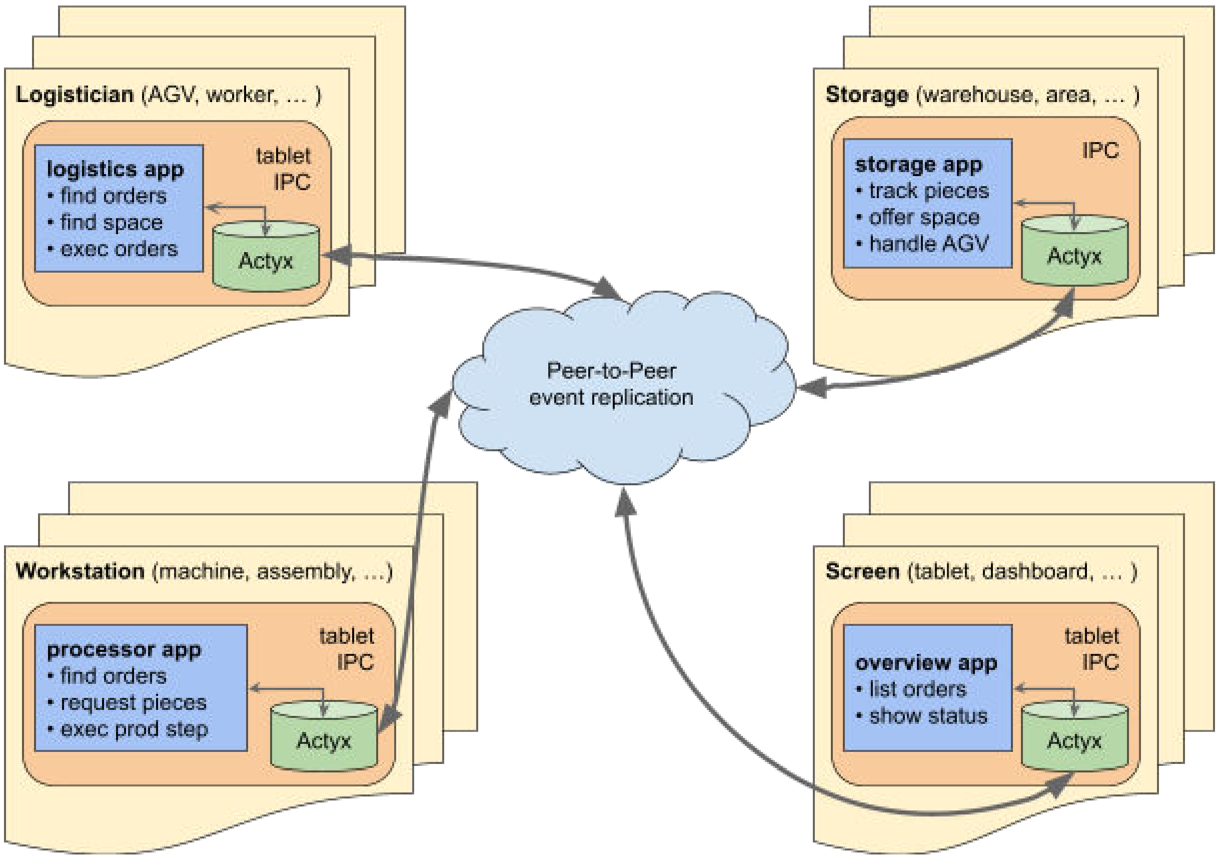

Figure 9: ACT baseline architecture and implementation

The above diagram illustrates the deployment structure and high-level architecture of the Actyx use case implementation. Each participant in the physical system (AGVs, machines, warehouses, workers, …)—called *agent* in the following—is represented as playing one of the four principal roles in this scenario (logistician, storage, workstation, screen). Every agent uses a computation device (a tablet or industry PC [IPC]) to participate in the system, running the Actyx middleware locally to serve the locally running apps; the diagram shows one such app per agent, but this is not meant to imply a limit: it is perfectly viable to run multiple apps on a device, accessing the same Actyx instance. The role of Actyx is to implement peer-to-peer event replication between computation devices without the need for other infrastructure—assuming that the devices can communicate with one another using TCP/IP.

While the Actyx middleware is an existing software product (implemented in the Rust language), all apps are written using the TypeScript language and the `@actyx/sdk` available in that language. There are no other restrictions, e.g. on UI frameworks, local state storage, ML libraries, etc.

2.4.2.1 Proof-of-concept stage

Before the beginning of TaRDIS the customer had already started implementing a proof of concept for the factory project described above based on the Actyx middleware and the Actyx Pond library, which offers a programming model very similar to a fully decentralised pub-sub system with the addition of an eventually consistent ordering of the received events—whenever events arrive late and need to be inserted at earlier times into the sequence, the application state is rewound to an earlier state and recomputed with the now more complete event sequence.

One drawback of this model is that the programmer needs to handle every possible application event—from the local node as well as remote nodes—at any time and in any sequence. This is necessary because some application state must be computed even from a currently incomplete event sequence while remote events are not yet fully replicated. The programmer must therefore encode not only the intended action sequence of a workflow but also all transient anomalies that can arise from uncoordinated swarm communication. While this is a burden in itself, it also adversely affects the communication between the programmer and the workflow designer (in this case a production expert with limited programming abilities): the structure of the code makes it difficult to judge whether the intended workflow is faithfully implemented.

Independent of the Actyx Pond application, work started on modelling the designed production lines in Siemens PlantSimulation, a tool for simulating the execution of production processes including intralogistics to assess whether the intended workflows work well together and estimate timings—this is crucial for assessing whether the design goal of finishing each production step within a single day is achievable, thus allowing all machining centers under construction to advance in lockstep every night. In addition, a visualisation was created for the Actyx Pond application to manually inspect whether the workflows function as expected. This was necessary not only to get an intuitive understanding of what was implemented, it was also crucial to understand whether the designed and implemented workflows were fit for purpose, i.e. whether their specifications were consistent and compatible. The Actyx Pond model only guarantees that all replicas of an entity eventually reach the same state, but offers no further guarantees regarding that state.

2.4.2.2 Baseline stage

In month five of TaRDIS the effort started to lift the proof of concept of the factory automation software to production level. This entailed a complete specification of the designed production workflows, a rigorous verification of their faithful implementation, handling of all error and failure cases, full test coverage of application state computation and progression, integration tests with multiple distributed participants to verify successful meshing of their various workflows, performance optimisations, the creation of debugging and operational tools, all accompanied by independent code reviews along the way. We discuss all these aspects in the following.

The specification of workflows has traditionally been done using PlantUML by that customer. In order to verify the equivalence of implemented code and specified diagrams, we created a pair of new Actyx libraries: “machine-runner”, accompanied by “machine-check”. The programmer uses the former for implementing the application logic in the shape of an event-driven state machine, then uses the latter for describing the designed workflow in a JSON format, verifying its well-formedness, and asserting that the implementation matches the specification. The JSON structure is very similar to the PlantUML text, making it easy to remove the possibility of miscommunication between programmers and production experts. This improvement is considered significant and extremely valuable by the customer, which is why we did not hold it back until the final use case implementation—the reason is that their distributed application will be used for real production in the factory later this year. In other words, the baseline implementation of the ACT use case employs Actyx tools that already include a TaRDIS improvement since it would have caused waste to withhold it.

Testing the application code is supported by the API design of machine-runner, which allows the test author to instantiate a machine in any desirable state, inject any sequence of events, and then verify that the expected state has been reached and all values have been computed correctly. Integration testing was done by running all parts of the application within a single process, replacing external hardware (like robots or AGVs) with emulators; integration between the application code and the AGV controller was tested by connecting to a dedicated hardware emulator that uses the original control electronics but contains electronic stubs in place of sensors and motors.

Once processes were working well enough, the usage of the Actyx middleware was optimised, cutting down waiting times of development–verification cycles considerably. Like querying a traditional SQL database, queries can be written more or less selectively, reading more or fewer events from storage; this is the main lever influencing how long each query takes to execute. After this it became obvious that the monitoring dashboard was causing high load on the Actyx database by polling for all workflows and entities at high frequency in order to update the screen with low latency—we fixed this by switching from regularly scheduled queries to standing subscriptions, a feature supported by Actyx in part thanks to TaRDIS. The next optimization was to cache machine-runner instances instead of recreating them for each UI update, which has a very similar effect (going from effectively polling to an event-driven subscription mechanism).

2.4.3 Baseline Interim Results

Since one of the goals of TaRDIS is to reduce the effort required for implementing correct heterogeneous swarm systems, we tracked in particular the effort expended by our customer for implementing the intralogistics workflows in the production setting (i.e. excluding the proof-of-concept state) and our effort in helping them achieve this, including code reviews.

The effort sums up to 7PM, where the resulting state is that a first complete set of workflows is fully implemented as described above, covering the first useful automation deployment to be rolled out to the factory (supplying tools and components to the production line, moving machining centers along the production line, moving finished machines to a storage area). This resulted in 2571 lines of code (LoC) written for the implementation of seven workflows, accompanied by 1447 LoC in protocol conformance and state computation tests.

2.4.4 Challenges

One main challenge for the implementation of the intralogistics workflows by our customer is that the factory is still being constructed and the production experts' design of the production processes is still evolving. This is typical for factory automation projects and a characteristic that needs to be taken into account when providing software tooling in this setting. The ability to match code to workflow designs using the machine-check library has already proved to be a step in the right direction, giving especially the project manager much higher confidence in the ability to successfully implement this project on budget and on time.

On the technical side there are a few issues with the programming model and Actyx APIs. On a conceptual level, it is still not straightforward and intuitive to consider the effects of uncoordinated swarm communication: a decision taken previously by one robot could be invalidated by a decision taken elsewhere, leading to the need of figuring out the scope of how much of the application state is now invalid and needs to be corrected. It would be highly desirable to get a clear indication from the underlying runtime system listing the invalidated events separately from the now valid ones.

Another issue is that even though the underlying state machine theory ensures deadlock freedom, the application can still get stuck in practice. The reason is that the theory assumes that if a robot has the possibility of performing a workflow action, it will eventually do so. However, it is difficult to enforce this in the TypeScript code of the application, e.g. due to exceptions prematurely terminating local computations. This and similar issues can also lead to resource leaks in the application, where state machines keep being updated with events received from the network without anyone listening or eventually shutting them down.

3 Delineation of TaRDIS Scope

3.1 Generic Requirements

This section enumerates properties that the TaRDIS toolbox must possess in order to be viable for implementing the use cases, but which do not go beyond the state of the art. We list them here because they cause effort when building the toolbox while not yielding measurable improvements, since implementing the use cases without them for comparison isn't meaningful.

Transport security: It is a baseline assumption today that network communication between computing devices must be secured by cryptographic means because the network that mediates the exchange cannot be trusted. This includes the authenticity of all messages (i.e. the recipient can verify the source), the integrity of all messages (i.e. that the recipient can detect whether messages have been modified in transit), and the confidentiality of all messages (i.e. that only the intended recipient can access the information contained in a message). The use cases do not require the property that the presence of communication cannot be detected by a third party—such protection requires undue messaging overhead for the cases we consider.

Transport timeliness: We do assume that given working network infrastructure the communication between peers is reasonably quick: interactions within the electric grid or a factory often proceed on a sub-second timescale. Consider for illustration that modelling the docking of a robot with a warehouse would not be viable if each message took a minute to be sent to the respective other party.

Support for hardware and OS: Another self-evident requirement is that the TaRDIS toolbox must be usable in the software development processes employed by the use case implementers. This pertains not only to the choice of programming language, but also includes that libraries and deployment components (like daemon processes) need to be compiled for the choice of hardware (i.e. CPU architecture) and operating system.

Execution on untrusted devices: The TaRDIS toolkit needs to be equipped to operate seamlessly on potentially untrusted devices, a crucial requirement given that certain scenarios will depend on end-user devices. Furthermore, it is imperative to operate under the assumption that these devices could be compromised. Consequently, TaRDIS will incorporate an identification system aimed at facilitating the identification and rectification of malfunctioning nodes, enhancing the overall security and reliability of software running within the toolkit.

3.2 Out of Scope for the Consortium

Several challenges brought up in the context of individual use cases go beyond the scope of work proposed for the TaRDIS toolbox or require tools and techniques that are not foreseen to be used within the consortium. In these cases we rely on the current state of the art.

Cryptography: Creating and integrating novel cryptographic primitives presents a challenge that lies beyond the scope of TaRDIS. When implementing TaRDIS cryptographic designs, we shall place reliance upon well-established industry-standard cryptographic libraries that have undergone rigorous testing, rather than resorting to self-implemented cryptography.

Peer-to-peer connectivity: A custom solution for the establishment of peer-to-peer network connections between different Internet subnets, commonly referred to as "hole punching", is also outside of scope. TaRDIS will instead leverage existing solutions, such as libp2p's excellent "hole punching" support in order to achieve the goal of seamless communication between different networks.

External system interfaces: One very common point—and frequently the most costly effort—when implementing a software project is to establish and implement the interfaces with external systems and components. TaRDIS does not aim to mitigate this issue or innovate in this space. Therefore, we will not consider such efforts when assessing the relative improvement achieved by using the TaRDIS toolbox; these efforts are considered out of scope.

Effect tracking in programming language: As was noted explicitly for the ACT use case, popular programming languages like TypeScript do not offer facilities for statically guaranteeing that the program will invoke a given function when it is in a given state. This has led to effective deadlocks in the baseline system even though the designed process was declared to be deadlock-free in theory. Similar issues will be encountered in the other use cases since e.g. the Java programming language has the same limitations. As these limitations are inherent to the chosen host language, TaRDIS cannot meaningfully innovate in this space given that our scope does not include the development of complete programming languages.

Establishing trust: Furthermore, ensuring that centralized services (e.g., cloud, network, content distribution providers, etc.) are inherently trustworthy, is beyond the scope of this project. Such resources can be considered to be reliable and trusted to execute the code placed in the hired resources without deviations, and with proper scalability and trustworthiness. In the case of protecting FL models by injection of differentially private noise, the trust model will be considered and defined precisely, when needed.

Satellite constellation operations: The communication scheduling in GMV´s use case is application specific. Unlike other use cases, in a constellation of satellites not all the nodes can communicate with each other at any time and their interactions are scheduled in advance according to the constellation geometry. TaRDIS will not address this very specific problem but will provide the right tools to let a space engineer develop specific functions to optimize this task while designing an application for a swarm of satellites.

4 Areas of Improvement

In this section we present the consolidated set of challenges identified during the use case baseline implementations, each one giving rise to an area of improvement targeted by the TaRDIS toolbox. We describe the improvement from an end-user perspective without prescribing details of the implementation, leaving those to the respective other work packages of this project.

Each area is associated with a set of key performance indicators [KPI] by which we will quantify the improvement achieved within TaRDIS. We describe the corresponding measurement techniques in a generic fashion, to be refined, detailed and first executed for report D7.2.

4.1 Membership and Communication

4.1.1 Overlay networks

Overlay networks are networks which are built on top of other networks, commonly over IP, or over other networks built on top of IP. These types of logical networks are of extreme importance for swarm systems as they function as the building blocks of most peer-to-peer communication. Choosing and implementing an overlay network protocol for a certain distributed system is a time consuming and specialized task.

TaRDIS will significantly improve on this issue, by offering an overlay network provider API where the programmer can select the properties of the desired overlay, and the overlay and implementation will be transparently selected for them. As an example, an overlay network may be used to disseminate information from one node to all others, from one node to another node, or from all nodes to all other nodes. By naming which property of these is desired, among others, the TaRDIS system will be able to select a proper protocol for the task.

KPI1: programmer effort for overlay

We measure this by

- counting the lines of code for implementing each overlay network, and
- counting the lines of code for unit testing the implementation

KPI2: network bandwidth used

We measure this by counting the bytes sent per second for

- overlay network formation
- overlay network maintenance
- user data transmission

Funded by the European Union

4.1.2 Nodes having sporadic connectivity

Feedback from developers involved with the baseline implementation strongly indicates that thinking in terms of uncoordinated swarm coordination is neither trivial nor intuitive: it takes significant effort to keep in mind that local information may be incomplete in many ways—not arbitrarily so, but the possibilities are numerous. When tackling this issue without tooling, the programmer needs to ubiquitously split program paths between expected messages and delays or failures. Since the number of cases to be handled ranges in the dozens even for very small workflows, this property of swarm systems significantly decreases developer confidence in their ability to deliver correct software.

TaRDIS will improve on this by providing a combined model of communication and computation that already includes uniform handling of the aforementioned vagaries of swarm systems. While it is not possible to completely hide the uncoordinated swarm nature without sacrificing partition tolerance or availability (cf. CAP theorem), we will provide an API that forces the programmer to provide the necessary reconciliation logic wherever conflicts can arise. We will quantify this improvement by asking programmers to assess their confidence level in the final use case implementation and comparing it to their confidence reached in the finished baseline implementation.

KPI3: programmer confidence

We measure this by sending a questionnaire to programmers.

KPI4: number of contingencies to be handled

We measure this by counting the code branches handling contingencies.

KPI5: delay caused by conflict resolution

We measure the time swarm participants spent in communicating and computing the result of conflict resolution.

4.2 Machine Learning

4.2.1 Splitting up a learning/inference problem for efficiency

Machine learning is known for its high potential for accuracy and efficiency improvement in prediction problems. All use cases plan to take advantage of it using TaRDIS, when dealing with complex tasks such as anomalies detections (ACT), orbit prediction (GMV), load forecasting and energy management (EDP), as well as cross-apps functionalities, for instance health monitoring and prediction (TID).

To improve the efficiency of machine learning/prediction models TaRDIS will provide a framework allowing the swarm nodes to share their local model coefficients/parameters based on locally processed data. Efficiency is improved by faster local processing and avoiding the need of transferring big amounts of data between the nodes.

KPI5: bandwidth consumption for training data

We count the bytes sent per second for transferring training data.

KPI6: FL CPU usage for training

We measure the percentage of CPU time spent on training ML models.

KPI7: FL training latency

We measure the time between starting training and obtaining a sufficiently converged ML model.

KPI8: FL storage/RAM requirements per node

We measure the minimum storage/RAM configuration under which training still succeeds.

4.2.2 Hierarchical federated learning

Primarily, federated learning (FL) was envisioned to operate in a client-server setting, with FL clients being either regular end-user devices (cross-device FL) or data/model servers (cross-sile FL). In either case, a centralized authority (FL server) orchestrates the FL process, by sending a global FL model to the participating clients, who then train the model on their local data before sending back their local model for aggregation from the server. However, recently it has become clear that, at least in the cross-device FL scenario, the scaling of the FL model training process to millions of devices can be a great challenge that remains to be solved. Furthermore, constructing the FL global model in a privacy-by-design fashion (i.e., in a way that users or the server cannot reverse engineer an individual user's data from their model), while maintaining high utility of the global FL model built can be a conflicting goal (e.g., high privacy means low utility of the model).

TaRDIS will improve on this issue by providing the necessary API and/or libraries that can be used by FL system developers (such as in the case of FLaaS) to build hierarchical FL solutions, that, in theory, have been shown to help on all three aforementioned fronts: 1) they can provide a better utility (e.g., test accuracy of the FL model closer to the one achieved at the centralized ML solutions), 2) they can relax the privacy requirement of trusting the server to aggregate and protect the client models, by allowing FL clients to trust an intermediary / hierarchical super-node instead, and 3) they can facilitate better the scalability of an FL solution to millions of end-user devices, since the intermediary / hierarchical layer of super-nodes can help distribute the network workload, and FL process management and maintenance.

KPI9: FL privacy

We will measure the 1) amount of privacy budget (epsilon) used during the ML model training, 2) the percentage inference gain of a membership inference attacker (we will also consider other types of attacks such as data reconstruction and attribute inference).

KPI10: FL accuracy

We will measure the ML model accuracy on a test set available at the FL server. Other metrics of model utility we are considering on the test set are AUCROC, F1-score, Precision, Recall, etc. These will be measured when comparing performance at the centralized vs. hierarchical vs. fully decentralised setting.

KPI11: scalability

We will measure the cost of maintaining each FL client connected with the FL server in the fully decentralized setting, and then the total number of FL clients that can be supported when introducing the hierarchical setup, and while keeping the overhead of communications between FL server and clients fixed.

4.3 Data Management

4.3.1 Partial replication of event logs

While the ACT use case already implements reliable event delivery to peers that can currently not be reached, the other use cases also assume that communication will eventually succeed. An obviously correct solution is to replicate all event logs to all peers whenever communication is possible; this is the scheme implemented in the Actyx middleware to date. The downside is that event logs occupy vastly more storage space on any given swarm participant that would be required for that participant's needs plus the redundancy needed to ensure reliable communication.

TaRDIS will improve on this by implementing a replication mechanism that ensures sufficient redundancy as well as delivering events to those swarm participants that need to process the contained information. The result is that local storage space requirements are no longer proportional to the size of the swarm, instead they scale with the number of peers a participant engages with through shared processes or workflows.

KPI12: data storage size needed per peer

We measure the disk usage on all edge devices used for storing event logs.

KPI13: latency at interested peers

We measure the time between event emission at one peer and event availability at another.

4.3.2 Replicating large pieces of data like FL models

The events that convey messages between swarm peers are typically small ($\lesssim$1kB) and can be transmitted quickly. Within the use cases there are pieces of data for which this is not true, e.g. manufacturing instructions ($\approx$100kB – 1MB) or ML models ($\approx$1 – 100MB). This information also needs to be shared efficiently and reliably with other peers. Current solutions involve either a central repository (which is not resilient) or replicate the data on all peers, leading to unnecessarily high storage size requirements.

TaRDIS will improve on this by implementing a replication mechanism that ensures sufficient redundancy as well as delivering large data to those swarm participants that need to process them.

KPI12: data storage size needed per peer

Funded by the European Union

We measure the disk usage on all edge devices used for storing large data.

KPI13: latency at interested peers

We measure the time between data storage at one peer and data availability at another.

4.4 Protocols

4.4.1 Conformance to design specified by non-programmers

In all use cases, the precise shape and function of swarm collaboration is defined by the respective domain experts (i.e. for power grid, smart homes, satellite constellations, factory operations, among others). These experts typically do not also specialise in software development, leading to additional effort as well as friction in the development process arising from the need to manually translate between the meaning of the written code and the process design supplied by the experts.

TaRDIS will improve on this by supplying a process design language that corresponds to a machine-interpretable software specification. The latter is tied to the offered TaRDIS APIs using each host language's native type system complemented by automatic protocol conformance testing to enforce that the software faithfully implements the process design. As a result, it will be possible for programmers and process experts to discuss the design and implementation using a common language. To make this process even more direct, there will be a graphical representation for the process design, employing the visual capabilities of the human brain for best results.

KPI14: non-conformance rate

We count the number of conformance defects found while using the software.

KPI15: programmer effort for conformance

We count the lines in conformance tests and the hours needed for achieving conformance.

KPI16: programmer & expert confidence

We measure this by sending a questionnaire to programmers.

4.4.2 Information flow security

In a heterogeneous swarm not all participants have the same access rights to information: events may be restricted to a selected set of participants as illustrated by the hierarchical treatment of federated learning training data in the TID use case. Processes designed as per the previous section need to be manually audited for violations of such constraints, which is tedious and error-prone.

TaRDIS will improve on this by offering a specification language for information flow security constraints, accompanied by tooling to analyse process designs for possible violations of these constraints. This will increase the confidence of domain experts in their process designs, and via the enforced protocol conformance this extends to increased confidence of

Funded by the European Union

programmers that their implemented software will comply with security requirements of the use case.

KPI17: security verification effort

We measure the number of hours it takes to verify security properties.

4.4.3 Verification of desirable properties

Not every process designed by domain experts guarantees proper function and desirable outcomes when executed under the particular constraints imposed by heterogeneous swarm systems. One example is that achieving (eventual) consensus on a process outcome is not trivial when taking into account that not all peers are authorized to see all information or are guaranteed to receive all information in the same order. This means that the designed process may not achieve what is intended.

TaRDIS will improve on this by providing analyses that allow designers to see whether a given process will work as they want. The formulation of the desirable properties to analyse for will require a deep understanding of the tools and techniques employed within the TaRDIS toolbox, but it will be easy for others (software developers or domain experts) to use the formulated properties in conjunction with the protocol design language mentioned previously in section 4.4.1.

KPI18: property verification effort

We measure the number of hours it takes to verify desirable properties.

KPI19: properties verified automatically

We enumerate (i.e. count) the properties that can be verified automatically.

4.5 Summary

The following table gives an overview of the KPIs and their applicability to the use cases.

Table 4: Overview of use case contributions to KPIs

KPI	EDP	TID	GMV	ACT
KPI1: programmer effort for overlay	✅			❓
KPI2: network bandwidth used				❓
KPI3: programmer confidence				
KPI4: number of contingencies to be handled				
KPI5: delay caused by conflict resolution		❓		
KPI6: FL CPU usage for training	❓		❓	
KPI7: FL training latency			❓	
KPI8: FL storage/RAM requirements per node				
KPI9: FL privacy				
KPI10: FL accuracy			❓	
KPI11: scalability		✅		
KPI12: data storage size needed per peer				
KPI13: latency at interested peers		❓		
KPI14: non-conformance rate				
KPI15: programmer effort for conformance				
KPI16: programmer & expert confidence				
KPI17: security verification effort	❓			
KPI18: property verification effort				✅
KPI19: properties verified automatically				✅

In the above means that we are reasonably certain that the given KPI can be measured both in the baseline and the final implementation of the respective use case. ❓ denotes KPIs where the feasibility of the measurement is being studied but not yet certain.

5 Conclusion

In this document we presented the implementation of the use case baselines, identifying the challenges that we intend to improve upon in the scope of the TaRDIS project by creating a toolbox of APIs that simplify the solution to the chosen use case problems. We found that these are aligned with the work plan of the other work packages. In particular, with the experience of the baseline implementations the use case partners were able to provide valuable insights shaping the design of emerging TaRDIS APIs, analyses, computational models, machine learning, communication, and data management facilities.

References

[1] Konecny, J., McMahan, H., Yu, F., Richtarik, P., Theertha Suresh, A., & Bacon, D. (2016). Federated Learning: Strategies for Improving Communication Efficiency. In the 29th Conference on Neural Information Processing Systems (NIPS).

[2] Geyer, R., Klein, T., & Nabi, M. (2017). Differentially private federated learning: A client level perspective. In NIPS Workshop: Machine Learning on the Phone and other Consumer Devices.

[3] Mo, F., Haddadi, H., Katevas, K., Marin, E., Perino, D., & Kourtellis, N. (2021). PPFL: Privacy-Preserving Federated Learning with Trusted Execution Environments. In Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (pp. 94–108). Association for Computing Machinery.

[4] "Amazon Web Services." Amazon Web Services, Year, https://aws.amazon.com/.

[5] "Google Cloud." Google Cloud, Year, https://cloud.google.com/.

[6] "Microsoft Azure." Microsoft Azure, Year, https://azure.microsoft.com/.

[7] "CIFAR-10 Dataset." CIFAR-10, Year, https://www.cs.toronto.edu/~kriz/cifar.html

[8] Varvello, M., Katevas, K., Plesa, M., Haddadi, H., & Livshits, B. (2019). BatteryLab, A Distributed Power Monitoring Platform For Mobile Devices. In Proceedings of the 18th ACM Workshop on Hot Topics in Networks (pp. 101–108).

[9] Schutz, Bob, Byron Tapley, and George H. Born. Statistical orbit determination. Elsevier, 2004.

[10] Rajan, John A. "Highlights of GPS II-R autonomous navigation." *Proceedings of the 58th annual meeting of the institute of navigation and CIGTF 21st Guidance Test Symposium (2002)*. 2002.

[11] Lv, Yifei, et al. "Evaluation of BDS-3 orbit determination strategies using ground-tracking and inter-satellite link observation." *Remote Sensing* 12.16 (2020): 2647.

[12] Wen, Yuanlan, et al. "Distributed orbit determination for global navigation satellite system with inter-satellite link." *Sensors* 19.5 (2019): 1031.

[13] Breen, Philip G., et al. "Newton versus the machine: solving the chaotic three-body problem using deep neural networks." *Monthly Notices of the Royal Astronomical Society* 494.2 (2020): 2465-2470.

[14] The Oxford Institute for Energy Studies, The electricity market design for decentralized flexibility sources, https://www.oxfordenergy.org/wpcms/wp-content/uploads/2019/08/↩ The-electricity-market-design-for-decentralized-flexibility-sources-EL36.pdf, 2019, Last accessed: 2023-09-29