H2020 – ICT-13-2018-2019

Machine Learning to Augment Shared Knowledge in Federated Privacy-Preserving Scenarios (MUSKETEER)

Grant No 824988

D7.1 Client connectors' architecture design – Initial version

November 19

Imprint

Contractual Date of Delivery to the EC: **30 November 2019**

Author(s):	**Susanna Bonura and Domenico Messina (ENG)**
Participant(s):	**ENG, IBM, IDSA**
Reviewer(s):	**Naoise Holohan (IBM), Joao Correia (B3D)**
Project:	**Machine learning to augment shared knowledge in federated privacy-preserving scenarios (MUSKETEER)**
Work package:	**WP7**
Dissemination level:	Public
Version:	**1.0**
Contact:	**Susanna Bonura – susanna.bonura@eng.it**
Website:	**www.MUSKETEER.eu**

Legal disclaimer

The project Machine Learning to Augment Shared Knowledge in Federated Privacy-Preserving Scenarios (MUSKETEER) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 824988. The sole responsibility for the content of this publication lies with the authors.

Copyright

Executive Summary

The client connector is the component required for a participant to join the MUSKETEER Platform. It is the software application supporting MUSKETEER platform participant in his data exchange, share and process, so to guarantee that he is sovereign of his data.

The client-side connectors have to support the set of privacy operation modes made available throughout the project according to the architecture defined in T3.1 and meet the requirements of the federated and privacy-preserving machine learning services designed in WP4. Moreover, the client component provides services for locally combining model updates into one consistent, up-to-date model instance. The client component serves as adaptor for the integration and industrial validation of the MUSKETEER platform in WP7.

The client connector, running on a secure and private space, provides the interface (Client Connector External APIs) for receiving a set of instructions from a master controller of the MUSKETEER server, related to the transferring of the required datasets and/or models (according to the POM chosen) from/to the MUSKETEER core to the secure and private space for the training of an MUSKETEER ML model.

It undertakes the responsibility of executing the received instructions with the use of the set of components that are running on the user's secure space, and comprises a machine learning environment which produces the local information updates (to be sent to the server) and incorporates global model updates (received from the server).

The client connector component is comprised of two local connectors. One is external, to allow users to share their (encrypted) data and/or to receive (encrypted) model updates generated on the server side, by exposing an endpoint to upload/download information and/or to retrieve trained models or (encrypted) model updates, depending on the Privacy Operation Mode.

The second one is local and implements a set of interfaces to access and, if needed, preprocess data stored in local databases or file systems.

Document History

Version	Date	Status	Author	Comment
0.1	16 September 2019	Table of Content	Susanna Bonura and Domenico Messina	First draft
0.2	20 September 2019	Architecture image added	Domenico Messina	Update
0.3	25 September 2019	Inputs for client connector general description	Susanna Bonura	Update
0.4	30 September 2019	Technical Requirements added	Susanna Bonura	Update
0.5	04 October 2019	Client Connector Architecture image updated and description added	Domenico Messina	Update
0.6	25 October 2019	For internal review	Susanna Bonura and Domenico Messina	Second draft
0.7	06 November 2019	Review inputs	Naoise Holohan	Update
0.8	21 November 2019	Review inputs	Joao Correia	Update
0.9	26 November 2019	Finalization	Susanna Bonura	Cleaning to be ready for submission
1.0	26 November 2019	Final Version	IBM	Final

Table of Contents

List of Figures

List of Acronyms and Abbreviations

Abbreviation	Definition
API	Application Programming Interface
CA	Consortium Agreement
DC	Data Connector
DP	Differential Privacy
DV	Data Value
FS	Feature Selection
FSM	Finite State Machine
GA	Grant Agreement
IDR	Intermediate Data Representation
IDS	Industrial Data Space
LC	Logistic Classifier
LGFS	Linear Greedy Feature Selection
MK	Master Key
ML	Machine Learning
MLP	Multi-Layer Perceptron
MN	Master Node
OS	Operating System
PERT	Program evaluation and review technique
PK	Public Key
POM	Privacy Operation Mode
PP	Privacy Preserving
PPML	Privacy Preserving Machine Learning
RAM	Reference Architecture Model
ROC	Receiver Operating Characteristics
SQL	Structured Query Language
TA	Task Alignment
UI	User Interface
WN	Worker Node

1 Introduction

1.1 Purpose

This document presents the first version of the MUSKETEER client connector architecture. It derives from technical requirements for MUSKETEER platform and from user needs of the two industrial scenarios considered within the project. The MUSKETEER client connector architecture is compliant with the general MUSKETEER platform architecture (presented in the deliverable D3.1 - Architecture design – Initial version). Both architectural designs (presented in the deliverables D7.1 and D3.1) are delivered at M12.

1.2 Related Documents

As already mentioned, the deliverable D7.1 - Client connectors' architecture design, is the first result of WP7 and in particular of the task T7.1.

The deliverable is the document describing the first version of main functionalities of the client connector. It contains the design of the component and how it interacts with services at server side.

For the design of the MUSKETEER Client Connectors presented in this document, the technical requirements described in the deliverable D2.1 and the overall architecture defined in the deliverable D3.1 are considered as main inputs (Figure 1).

Finally, together with the deliverable D3.1, the D7.1 constitutes a basis for the remaining deliverables of the work packages regarding architecture, development, testing and validation of the MUSKETEER platform.

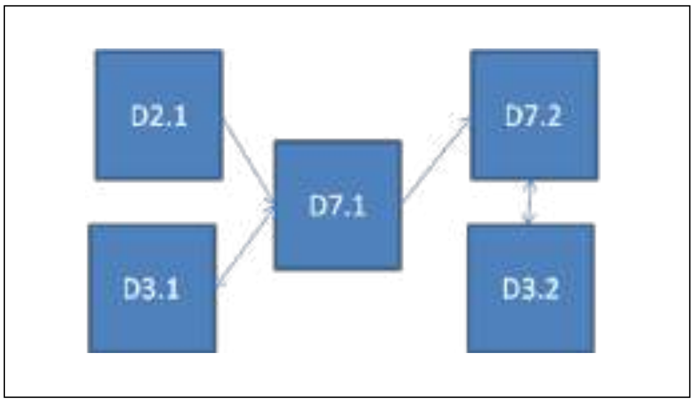

Figure 1 – Related documents

1.3 Document Structure

In the next section (Section 2), the general description of client connectors is presented according to the IDSA connector specifications.

In Section 3, MUSKETEER platform technical requirements are recapped in order to cross check that the requirements involving end user software side, are met in the client connector architecture design.

In Section 4, a summary of the MUSKETEER platform architecture is presented, so to have a comprehensive picture before detailing client connectors.

Section 5 presents the first version of the MUSKETEER client connector architecture which is the first results of the task T7.1

Finally, Section 6 concludes the deliverable. It outlines the main findings of the deliverable which will guide the future research and technological efforts of the consortium.

2 Client Connector: a general description

The client connector is the component required for a participant to join the MUSKETEER Platform. In general, connectors are the central technological building block of the International Data Spaces [1]. They are software components supporting participants in their data exchange, share and process. At the same time, connectors guarantee that the Data Owner is sovereign of his data.

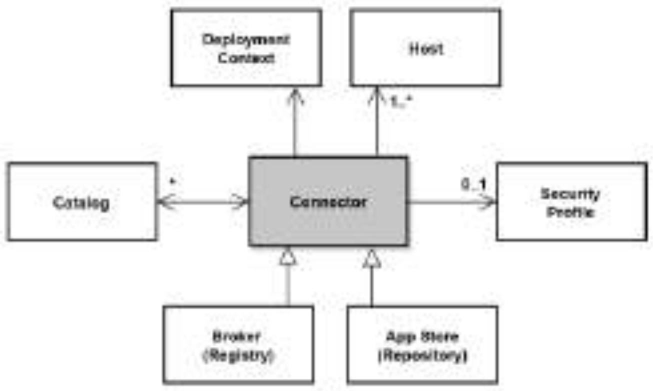

Figure 2 – Connector concept [1]

Figure 2 shows the main elements composing a connector. The deployment context of a connector records the connector's location, as for example the data center and coordinates, the type of its deployment (on-premises or cloud-based), and the name of the Participant.

The security profile indicates the capabilities of a connector to maintain a controlled, secure and trusted environment for exchanging, sharing and processing data, through remote integrity verification, application isolation, usage control support.

The catalog represents the repository of the metadata of resources, constructed in accordance with the IDS Ontology, through which connectors provide or consume data.

Optionally, the catalogue, or individual sets of resource metadata, may be advertised via intermediary nodes, such as the broker service provider, which is an intermediary that stores and manages information about the data sources available in the International Data Spaces, or in the app store.

Each host represents an individual communication capability of the connector, a server that exposes resources via endpoints (HTTPS URLs, MQTT topics, etc.) according to the communication protocol supported [1].

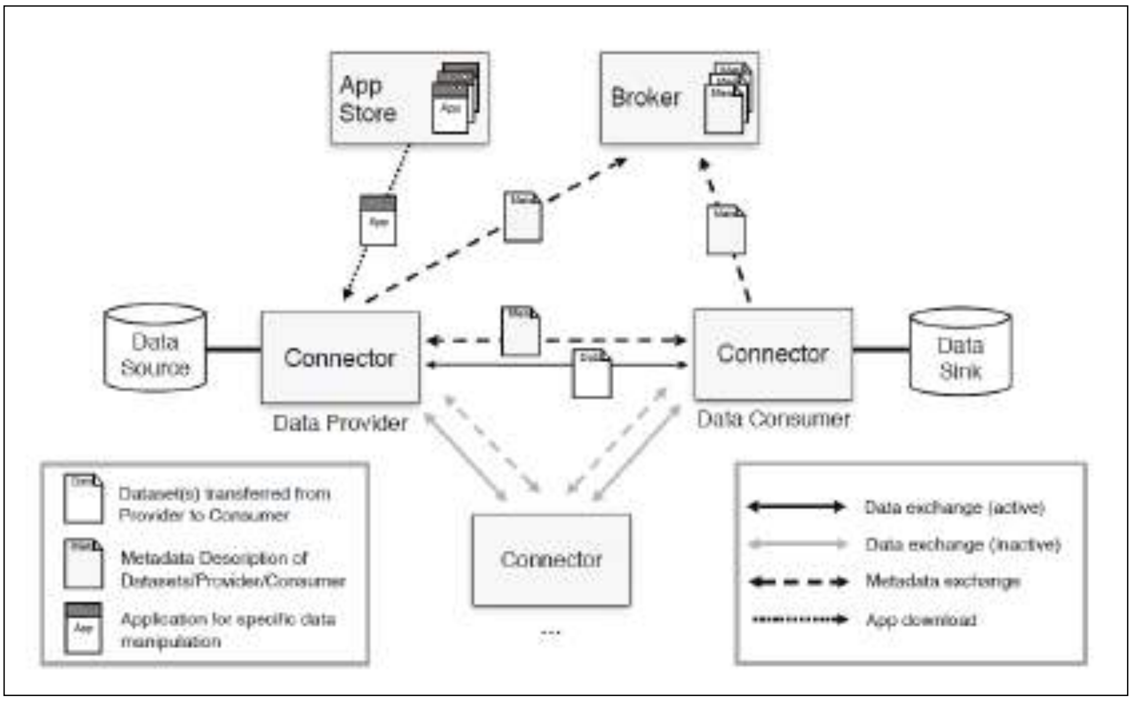

Figure 3 – Interaction among connectors according to IDSA RAM [1]

According the IDSA Reference Architecture Model (RAM), in general, a set of applications (Apps) are deployed inside the connector, to facilitate data processing workflows.

The Figure 3 shows interaction among connectors according to IDSA RAM.

Every connector participating in the IDS Platform must have a unique identifier and a valid certificate. In addition, it must check and verify the identity of other connectors and provide a valid certificate, so that each participant in the platform is able to verify the identity of any other participant. The connector serving as the data source must be able to verify the receiving connector's capabilities and security features as well as its identity.

Communications among connectors can be encrypted and integrity protected. In addition, connectors must be able to ensure that IDS participants' data is handled according to the usage policies specified: otherwise the data will not be exchanged.

Data providers and data consumers must be able to decide which level of security they intend to apply for their respective connectors by deploying connectors supporting the selected security profile.

Participants must have the opportunity to describe, publish, maintain and manage different versions of metadata, which describe the syntax, serialization and semantics of data sources.

The operator of a connector must be able to provide an interface for data and metadata access, to transmit metadata of its data sources to one or more brokers, and to browse and search metadata in the metadata repository, provided the participant has the right to access the metadata.

To create and structure metadata, the operator of a connector may use vocabularies, that can be already existing or created ad hoc by the operator.

Vocabulary hubs are central servers that store vocabularies and enable collaboration, searching, selection, matching, updating, requests for changes, version management, deletion, duplicate identification, and unused vocabularies.

Participants should be able to run the Connector software in their own IT environment.

Alternatively, they can run a Connector on mobile or embedded devices. The operator of the Connector must be able to define the data workflow inside the Connector. Users of the Connector must be identifiable and manageable. Passwords and key storage must be protected. Every action, data access, data transmission, incident, etc. should be logged. Using this logging data, it should be possible to draw up statistical evaluations on data usage etc. Notifications about incidents should be sent automatically.

The Connector receives data from an enterprise backend system, either through a push-mechanism or a pull-mechanism. The data can be provided via an interface or pushed directly to other participants.

Finally, according the IDSA RAM, other Connectors can subscribe to data sources or pull data from these sources. Data can be written into the backend system of other participants.

3 MUSKETEER Technical Requirements

This section shows the list of the technical requirements that were elicited starting from the business requirements coming from the description of the user stories in Smart Manufacturing and Health scenarios (for more detail please see the deliverable D2.1 - Industrial and Technical Requirements).

In the section 5, the MUSKETEER Client Connector architecture is presented so to meet the technical requirements involving end user IDS operator.

The list of such technical requirements is as follows:

TR001 The MUSKETEER platform shall ensure that access control over datasets is applied according to the data policies and the terms of relevant active valid data sharing contracts.

TR002 The MUSKETEEER platform shall forbid unauthorised user access to the platform and the datasets.

TR003 The MUSKETEEER platform ensures different authorisation levels for accessing datasets.

TR005 MUSKETEEER end user must have a unique identification that will be used in all the data exchange/communications.

TR006 Registration into the MUSKETEER Platform with username a password.

TR007 MUSKETEEER end user could create one or more new tasks.

TR008 In MUSKETEER a task must be defined as a problem statement that feeds from data and produces a trained machine learning model.

TR009 New tasks should obtain unique task identifier.

TR010 In MUSKETEER a task should have a general description.

TR011 The MUSKETEER Platform must, for each task, uniquely identify every input data from every end user.

TR012 Description of the input features. The meaning of every field must be explicitly described.

TR013 The MUSKETEER Platform must share a set of pre-processing algorithms such that every end user pre-processes its own raw data to obtain a common representation (e.g. high pass filtering, edge detection, bag of words with TFIDF weighting …).

TR014 MUSKETEER pre-processing modules should always produce an output vector with the expected content and format.

TR015 MUSKETEER must make any ad hoc pre-processing algorithms (defined and implemented by the end users) shared with other users contributing to the task.

TR016 In MUSKETEER, for each task, definition and nature of the problem to be solved, must be shared among all the participants in a task such that they can contribute with new data to the training process.

TR017 MUSKETEER Privacy Operation Modes must cover all kinds of privacy restrictions that end users would apply to his/her data.

TR018 Privacy restriction should be described in natural language to facilitate the specification of the task to the end user.

TR019 MUSKETEER Platform should envisage monetary rewards as well as collaborative results.

TR020 Browsing for published active tasks by MUSKETEER end users.

TR021 Creation and/or access to published active tasks by MUSKETEER end users.

TR022 Running of the training procedure associated to a given MUSKETEER ML task.

TR023 Monitoring of the progress of MUSKETEER ML tasks until completion.

TR024 MUSKETEER must provide the outcome of a task (reward, trained model, etc.).

TR025 MUSKETEER must allow data to be transferred and joined either in the server or in a given user.

TR026 MUSKETEER must support the case where no raw data is transferred outside the client facilities (the ML model training must take place in the server by using the aggregated information from the clients).

4 MUSKETEER Platform Architecture

The MUSKETEER platform must provide the infrastructure and implement the services that are required to enable the distributed machine learning capabilities developed in WP4 and WP5, along with interfaces supporting the use case integration in WP7.

The first version of the MUSKETEER platform architecture is described in the deliverable D3.1. It is based on a micro-services architecture and is designed to meet the security requirements of industrial data standards. Furthermore, it ensures the scalability in the number of users, data volume and complexity of the machine learning models [2].

The MUSKETEER platform architecture is client-server (see Figure 4): the server component coordinates the secure transportation of information and models across the participants, the client components allow use case participants to share information (the kind of data to be exchanged will depend on the Privacy Operation Mode), retrieve model updates, incorporate those locally, and ultimately retrieve the trained machine learning models for the deployment in their local business processes.

The server infrastructure is provided by IBM, using the IBM® Cloud™ platform. It will host micro-services for data management, machine learning and internal/external data exchange (i.e. IBM Cloud™ Messages for RabbitMQ, IBM® Db2® on Cloud, IBM® Cloud Object Storage, IBM Cloud™ Functions, IBM Cloud™ Kubernetes Service).

For data management, different database persistence layers are envisioned to support different types of data, like Object Storage for large unstructured data (such as images), or Relational Database Management Systems for relational data (such as numerical sensor measurements and associated metadata).

With regard to the machine learning capabilities, algorithms provided will be based on Python with deep learning frameworks such as TensorFlow [1], Caffe[2], PyTorch[3] etc. to support efficient execution of model training on CPU and GPU processing.

An external connector implements the features that allow users to upload/download information and/or to retrieve trained models or (encrypted) model updates, depending on the Privacy Operation Mode.

Security measures envisioned on the server side include at-rest and in-flight data encryption, integration of identity and access management, and Bring-Your-Own-Key (BYOK) encryption.

As mentioned, the detailed MUSKETEER platform architecture design can be found in the document D3.1.

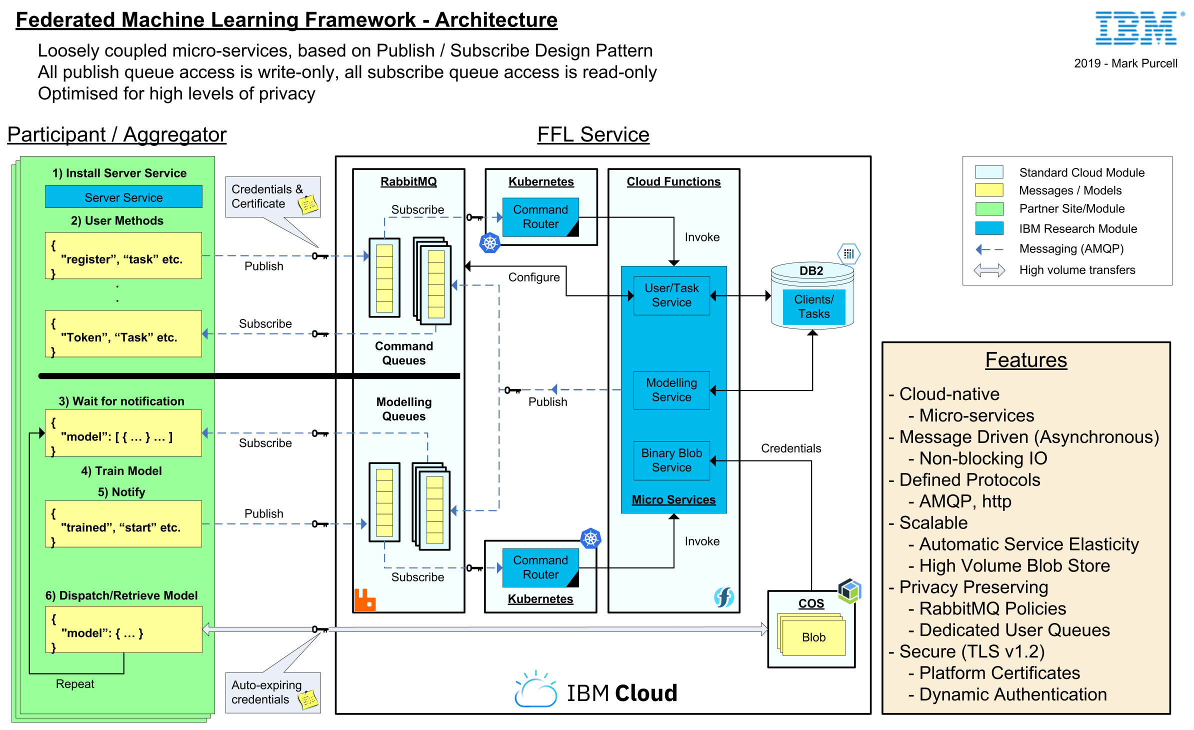

Figure 4 – MUSKETEER Platform Architecture

[1] https://www.tensorflow.org/

[2] http://caffe.berkeleyvision.org/

[3] https://pytorch.org/

Also the client component architecture is based on micro-service approach and comprises a machine learning environment which produces the local information updates (to be sent to the server) and incorporates global model updates (received from the server). It is further detailed in Section 5.

For the communication between the client and server components standard protocols such as HTTPS and AMQP are exploited with particular emphasis on Transport Layer Security. The architecture has to support asynchronous communication, in particular instances where a given client fails to produce model updates, which should not result in a blocking of the machine learning process for the remaining participants.

5 MUSKETEER Client Connector Architecture

As already mentioned, the MUSKETEER platform is a client-server architecture, where the client is intended as a software application that in general is installed on-premise and run at every end user side. Such a software application is named client connector in the MUSKETEER taxonomy.

On the other hand, the MUSKETEER server is the central part of the platform that communicates with all the client connectors and acts as a coordinator for all the operations. Users federated to MUSKETEER, interact with the client connector installed on their side and that client will communicate with the server to perform several actions on the platform.

It is worth mentioning that all technical requirements, as they are described in the deliverable D2.1 are met in the MUSKETEER architecture (both client and server side); but such requirement list will be further revised, refined and improved during the project execution; as a consequence also the general MUSKETEER architecture and the client connector architecture will be updated, respectively in the deliverables D3.2 and D7.2, so to meet all the technical requirements elicitated.

For the aims of the present document, only the technical requirements involved in the client side will be reported in round brackets.

The interactions between the MUSKETEER server and the client connector aim at supporting two types of activities: the first one is operative and regards interactions to exchange messages and pieces of information for the general operation of the platform (create a new user, 'log-in' as a user, define a task, declare privacy preferences, describe data attributes, feedback reporting, etc.); the second one regards the actual model training, that is the interaction to exchange the messages and pieces of information during the training phase of the ML algorithm declared in a task.

A ML task may be as a problem statement that feeds from data and produces a trained machine learning model as an outcome. Any MUSKETEER end user can create one or more new tasks thereby obtaining a unique task identifier (task_id) from the platform and run them even in parallel.

The definition step requires the specification of the task characteristics:

- General description: a high-level description of the task (the problem to be solved) is necessary for a rapid identification of existing tasks by other users. In a first version of MUSKETEER, this part can be reduced to a minimum, since the end users and their tasks are already defined.
- Data: it is recommended to facilitate an initial dataset, i.e., some data illustrating the task to be solved. Data comprises input feature vectors ($\underline{x}$) (for non-supervised tasks) and pairs of input feature vectors and target values ($\underline{x}$, t) (for supervised tasks).
- Features description: a general description of the input features (like defining the fields in a Table) is necessary to unify the data representation among users and finally being able to combine all the contributed data during the learning stage. In the above-mentioned general case where input data is represented as a vector, the meaning of every field in such a vector must be explicitly described, for compatibility purposes.
- "Ad hoc" preprocessing algorithm (optional): in some cases, input data is not so easily represented in terms of individual meaningful features and the feature vector may be the result of applying some (possibly complex) preprocessing to a raw piece of information (for instance an image, a text, a voice recording, etc.). In those cases, when the raw data cannot be transmitted to the MUSKETEER server and processed in the same place, it is necessary to share the preprocessing algorithm such that every end user preprocess its own raw data to obtain a common representation into the feature vector x. For instance, in the image case, some transformations may need to be applied to an image before feeding it into a machine learning model, such as high pass filtering followed by an edge detection, a specific feature extraction, etc.; in the case of texts, the preprocessing could be a bag of words with TFIDF weighting, etc. The casuistic can be extremely large and problem dependant, so it is important to guarantee that the preprocessing module always produces an output vector with the expected content and format and it is recommended that the "ad hoc" (non-standard) preprocessing algorithms be defined and implemented by the end users defining a specific task, such that if can be shared with other users contributing to the task as a "preprocessing object".
- Target values: the problem to be solved is defined by the target values (in supervised tasks). For instance, given the above described features, the target could be to esti-

mate the annual income in euros, or to estimate if that person is unemployed or not. The definition and nature of the target must also be shared among all the participants in a task such that they can contribute with new pairs (x, t) to the training process.

- Privacy requirements: it is important that the end user determines which are the privacy restrictions that apply to his/her data, because those restrictions will determine the POMs (Privacy Operation Modes) that can be used. It is more operative that the user declares the privacy restrictions that apply to the data and then the platform offers the available POMs to solve the ML task. The 'data_privacy' parameter can be chosen among several options, described in natural language to facilitate the specification of the task to the end user, for example, one end user may adhere to some of the following statements:
  - my data is open and can be freely distributed;
  - my data can be shared after anonymization;
  - my data can be shared only with the MUSKETEER platform under some confidentiality agreement with the platform;
  - my data can be shared with other end users under some confidentiality agreement with the end users;
  - my raw data cannot leave my facilities (only the MUSKETEER client can see it and obtain some operations on it: gradients, dot products, etc., but never reveal individual data points.)
  - my data can be used for a given task, but not for other tasks.
- Reward: this is the motivation for other users to join the task and offer their own data. After completion of the task, the reward is shared among the participants. In the context of this project the motivation by the end users is taken for granted, but it is important not to forget its existence, specially to motivate the data value estimation procedures to be developed in WP6.

  The expected reward could be, for example, among other possible options are:
  - monetary: a user wants to improve a ML model, he/she deploys a MUSKETEER task and offers a monetary reward upon successful completion of the task. Other users owning data of potential interest for the task can join the initiative. After training is completed, the contribution of every participant will be (hopefully) determined by the data value estimation modules, and the reward will be distributed according to the value of every contribution. The initial user retains the trained model for its own use.
  - collaborative results: all participants contribute with a portion of data to the task and the resulting learned model is shared among all participants in the task (well, possibly only among those identified as positively contributing to the task solution).

Hence, once in the platform, any end user will have mechanisms to:

- browse for published active tasks, and join one or more of them
- create his/her own task as explained above
- run the training procedure associated to a given ML task and follow the progress until completion
- receive the outcome of a task (reward, trained model, etc.)

From a technical point of view, the client connector provides metadata as specified in the connector self-description, e.g. technical interface description, authentication mechanism, exposed data sources, and associated data usage policies. Thus, metadata is a fundamental building block for the deployment and composition of several ML model definitions inside a client connector: all operations are defined in terms of input and output parameters, bound protocols, and endpoints. Preconditions and postconditions need to be made explicit, and effects on the environment must be outlined.

Indeed, according the IDSA Reference Architecture Model (RAM), for each client connector, it is possible to specify pre- or post-conditions that have to hold before (as the integrity check of the environment) and after (as the data item is deleted after usage) decision-making. In addition, it is possible to define on-conditions that have to hold during usage (e.g., only during business hours). These conditions usually specify constraints and permissions that have to be fulfilled before, during, and after using data [1].

The client connector must allow participant to share information (the kind of data to be exchanged will depend on the Privacy Operation Mode), retrieve model updates, incorporate those locally, and ultimately retrieve the trained machine learning models for the deployment in the local business processes.

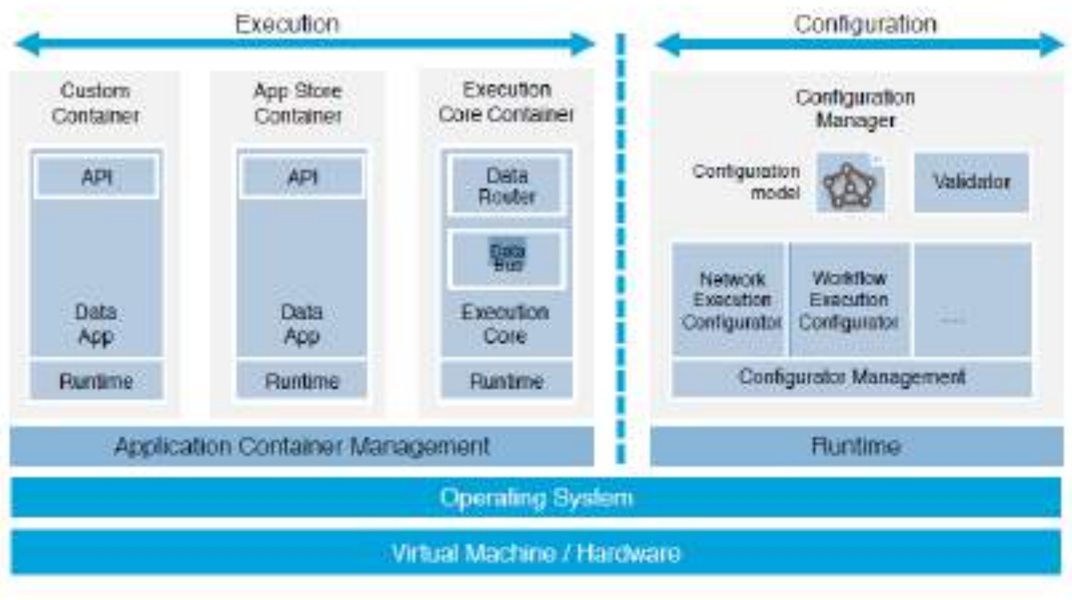

Figure 5 – Connector system layer architecture according to IDSA RAM [1]

The connector architecture, as described within the IDSA RAM, uses application container management technology to ensure an isolated and secure environment for individual data services.

The Figure 5 shows the connector system layer architecture presented in the IDSA RAM as splitted in two phases: execution and configuration.

Within the execution phase, we have six entities involved. The first one is the Application Container Management component. It supports the deployment of an Execution Core Container and selected Data Services. Thereby, Data Services are isolated from each other by containers so to prevent unwanted interdependencies. Using Application Container Management, it is possible to apply extended control of Data Services and containers.

The second element involved in the execution phase is the Execution Core Container, which provides components for interfacing with Data Services and supporting communication. More in detail, within an Execution Core Container, a Data Router handles communication with Data Services to be invoked according to predefined configuration parameters. In this regard, it is responsible for the way the data is sent (and received) to (and from) the data bus by (and to) Data Services. Participants have the option to replace the Data Router component by alternative implementations of various vendors. If a connector in a limited or built-in platform consists of a single data service or a fixed connection configuration (eg. On a sensor device), the data router can be replaced by hard-coded software or the data service can be exposed directly. The Data Router invokes relevant components for the enforcement of Usage Policies, as configured in the connector or specified in the Usage Policy.

In addition, within an Execution Core Container, the Data Bus is present, to exchange data with Data Services and Data Bus components of other Connectors and also to store data within a Connector. In general, the Data Bus provides the method to exchange data among Connectors. Like the Data Router, the Data Bus can be replaced by alternative implementations in order to meet the requirements of the operator. Like the data router, the data bus can be replaced by alternative implementations in order to meet the operator's requirements.

The third element involved in the execution phase is the App Store Container (one for each Data Service), which is a certified container downloaded from the App Store and provides a specific Data Service to the Connector.

The fourth element involved in the execution phase is the Custom Container, which provides a self-developed Data Service. Custom containers usually require no certification.

The fifth element involved in the execution phase is the Data Service, which defines a public API, which is invoked from a Data Router. A meta-description specifies this API and is imported into the configuration model. Data Services can be implemented in any programming language and target different runtime environments. The tasks to be executed by Data Services may be different. Existing components can be reused to simplify migration from other integration platforms.

Finally, the Runtime of a Data Service is found in the execution phase. It depends on the selected technology and programming language. The Runtime and the Data Service represent the main part of a container. Different containers may use different runtimes. What runtimes are available depends only on the base operating system of the host computer. From the runtimes available, a service architect may select the one deemed most suitable.

With regard to the configuration phase, the connector architecture envisages five elements.

The first one is the Configuration Manager, which represents the administrative part of a Connector and it is in charge of managing and validating the Configuration Model, followed by deployment of the Connector. Deployment is delegated to a collection of Execution Configurators by the Configurator Management.

The second element is the configuration model, which is an extensible domain model to describe the configuration of a connector. It consists of configuration aspects that are interconnected and independent of technology.

The third element is the Configurator Management. Its main task is to load and manage an exchangeable set of Execution Configurators. When a Connector is deployed, the Configurator Management component delegates each task to a special Execution Configurator.

This last one is the fourth element. Execution configurators are interchangeable plug-ins that perform or translate individual aspects of the configuration model into a specific technology. The procedure for performing a configuration depends on the technology used. Common examples could be the generation of configuration files or the use of a configuration API. By using different execution configurators, new or alternative technologies can be adopted and integrated into a connector. Therefore, every technology (operating system, application container management, etc.) gets its own execution configurator.

The last element present in the Configuration phase of a Connector, is the Validator, which is in charge of checking if the Configuration Model complies with self-defined rules and with general rules specified by the International Data Spaces, respectively. The violation of the rules can be considered as a warning or an error. If such warnings or errors occur, the deployment may fail or be rejected.

Since the configuration phase and the execution phase are separated from each other, it is possible to develop and subsequently operate these components independently of each other.

According to the IDSA RAM, different implementations of the connector can use various types of communication and encryption technologies, depending on the requirements indicated.

The functional architecture of the MUSKETEER client connector is described in Figure 6.

The MUSKETEER Client Connector enables the user to interact with the MUSKETEER Server so to take part to the Federated Machine Learning processes, according to the specifications defined in the project. In order to be compliant with the IDSA RAM [1] shown so far, the client connector will be released as a multi-container application that will support both on-premise distributed environments as well as cloud providers.

Containers allow to run an application and all of its dependencies in isolated processes. The goal of containerization is to allow to easily package everything is needed to run software reliably when moved from one environment to another. There are a number of benefits that moving to containerization provides. Some of the main benefits companies can see include increased portability, simple and fast deployment, enhanced productivity, possible lower cost, improved scalability, improved security (**TR002**).

For the correct usage of the access control mechanism the design of an effective user management process is envisaged (**TR001**, **TR003**) so that each client connector will be univocally identified (**TR005**). In MUSKETEER platform, a potential approach will be to include two main subprocesses: a) the registration and subsequently user creation (**TR006**), and b) the user authentication (login) that enables the access to the platform as a whole. In MUSKETEER platform, the users appear under the concept of organisations. The registration process is handled by a component and includes the following steps:

a) The organisation manager submits the organisation signup form.

b) The MUSKETEER administrator receives the request, checks and approves it.

c) The organisation manager creates the invitations for the organisation members. Each member receives the invitation link via email accompanied with an invitation token.

d) Each organisation member fills-in the member signup form providing also the invitation token and, upon successful registration, access is granted to the MUSKETEER platform.

A further analysis and validation will be performed before outlining the final approach in the final version of the deliverables on the MUSKETEER Platform and Client Connector Architecture (D3.2 and D7.2)

The client connector architecture implements an intuitive user interface through which the user will be able to perform the canonical operations of the MUSKETERR platform, such as browsing published active tasks (**TR020**), joining one or more of them, creating her own task (**TR007**, **TR021)**, running the training procedure associated to a given ML task (**TR022**) and following the progress until completion (**TR023**), receiving the outcome of a task (reward, trained model, etc) (**TR019**, **TR024**).

The user interface will allow end users to straightforwardly define a task (**TR008**, **TR010**, **TR012**) that will be univocally determined (**TR009**, **TR011**)

The communication with the MUSKETEER server will occur through the MUSKETEER Client Processor module that implements the external connector exploiting the MUSKETEER-Client python APIs provided by IBM[4].

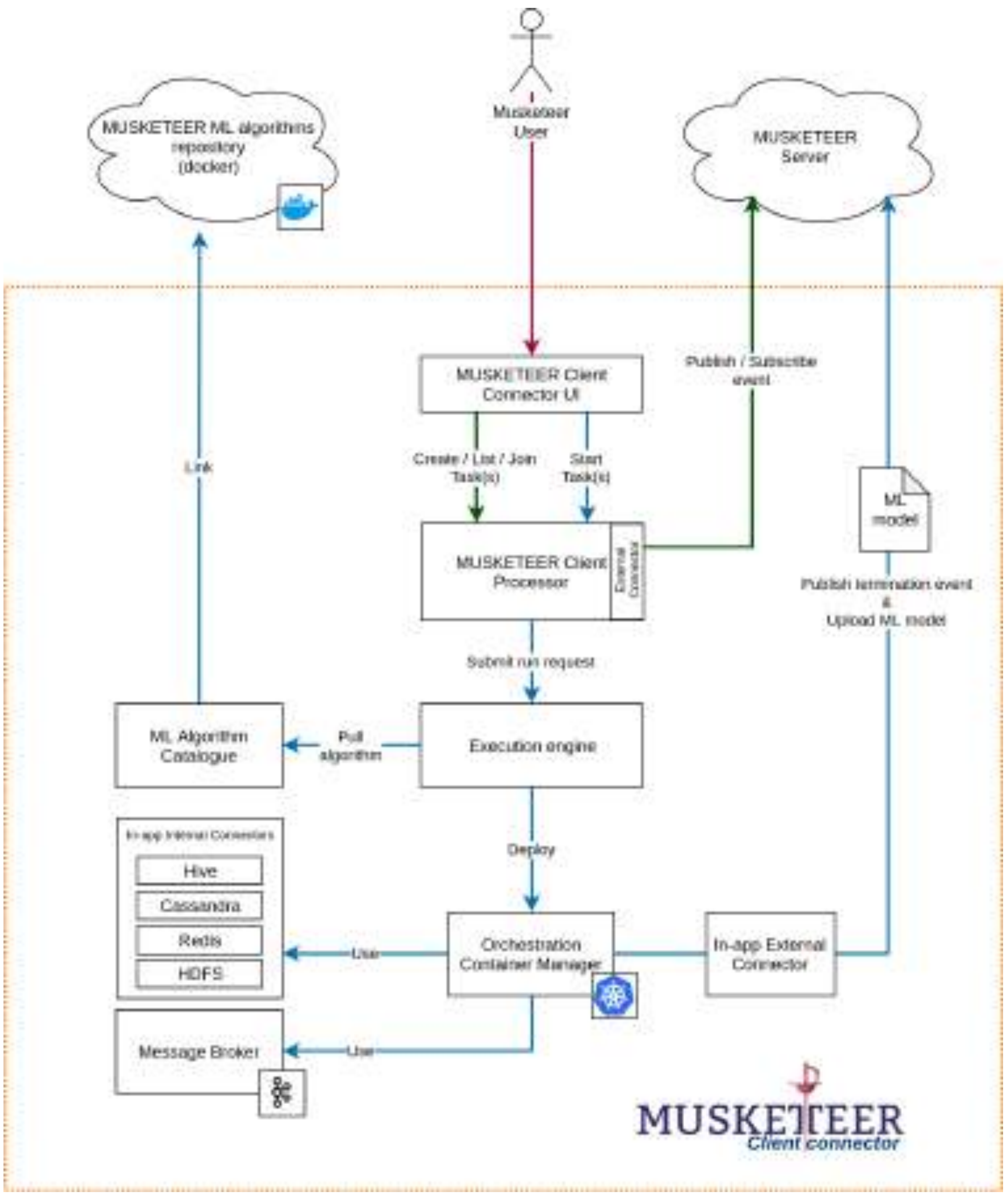
MUSKETEER ML algorithms repository (docker)
Musketeer User
MUSKETEER Server
MUSKETEER Client Connector UI
Publish / Subscribe event
Link
Create / List / Join Task(s)
Start Task(s)
ML model
MUSKETEER Client Processor
External Connector
Publish termination event & Upload ML model
Submit run request
ML Algorithm Catalogue
Pull algorithm
Execution engine
In-app Internal Connectors
Hive
Cassandra
Redis
HDFS
Deploy
Use
Orchestration Container Manager
In-app External Connector
Message Broker
Use
MUSKETEER Client connector

Figure 6 – MUSKETEER Client Connector Architecture

[4] https://github.com/IBM/Musketeer-Client

Since the client processor is shipped as a docker image, it will run in a sandboxed execution context and will securely communicate through the external connector with the server, so that the compliancy with the IDSA RAM [1] will be preserved in all the parts of the Client Connector.

In order to run the algorithms in a safe and isolated environment, the execution engine provides all the capabilities to manage the lifecycle of the running jobs locally. Such an execution engine has to support the deployment, execution, monitoring and orchestration of algorithms as micro-service both in streaming and batch mode.

The chosen ML algorithm micro-services are retrieved from the ML algorithm catalogue and are instantiated according to the resources available on the machines in which the Client Connector runs.

The ML algorithm catalogue gathers all the machine learning models created in the project to cover a variety of privacy-preserving scenarios and ensure security and robustness against external and internal threats (**TR017**, **TR018**).

More in detail, the library will contain a complete set of algorithms for data pre-processing, normalization and alignment of horizontal and vertical distributed datasets (**TR013**, **TR014**, **TR015**); models for data value estimation; supervised learning algorithms to solve regression and classification tasks (Linear models like Logistic regression or ElasticNet, Kernel Methods such as semiparametric SVMs, Tree Based Algorithms such as Random Forest and Deep Neural Networks such as MLPs or CNNs); unsupervised learning to perform clustering or topic modelling (methods like K-means or LDA). Such pre-processing and training algorithms will run under different privacy operation modes (**TR016**, **TR025**, **TR026**) in which the platform can operate.

it is worth mentioning that in order to cover the largest possible number of industrial scenarios, MUSKETEER has to support several POMs. The main features to compare these POMs are the following ones:

- Privacy level: This is possibly the most obvious requirement in any IDP where data is to be shared.
- Computational local overload: Some problems require standard computational means, while in other cases, special computational resources might be needed: a Spark cluster or GPU units, for instance.
- Central Storage requirements: This requirement is mainly to be fulfilled by the central platform; it is needed if the users' data is collected and stored in a single place (a cloud service, for instance).

- Communication requirements: Depending on the volume of the datasets and the type of machine learning algorithm, large communication resources may be needed.
- Data Utility Accountability: It is important to correctly evaluate the relevance/contribution of the provided data for the resulting final machine learning model.

Each ML algorithm micro-service will be packed as Docker image. A Docker image is an artifact used to execute some software in a Docker container. An image is essentially built from the instructions for a complete and executable version of an application, which relies on the host OS kernel. When the Docker user runs an image, it becomes one or multiple instances of that container.

Docker is an open source OS-level virtualization software platform primarily designed for Linux and Windows. Docker uses resource isolation features of the OS kernel, such as c-groups in Linux, to run multiple independent containers on the same OS. A container that moves from one Docker environment to another with the same OS will work without changes, because the image includes all of the dependencies needed to execute the code[5].

A container differs from a virtual machine (VM), which encapsulates an entire OS with the executable code atop an abstraction layer from the physical hardware resources.

Within the end user's virtual machines dedicated to run the client connector, resources are supervised by the Orchestration Container Manager which is the component that provisions the runtime environments for each ML algorithm.

As Orchestration Container Manager, Kubernetes was chosen. It is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery [6].

It is worth noticing that there's a perfect match with the concept of Custom Container and/or App Store Container in the IDSA's RAM and, as an internal communication mechanism, a message broker will be used so that the Client Connector can coordinate properly the workflows and the dataflows.

Such a message broker may be based on Kafka, which allow to publish and subscribe to streams of records, similar to a message queue or enterprise messaging system, store streams of records in a fault-tolerant durable way and process streams of records as they occur.

[5] https://www.docker.com/

[6] https://kubernetes.io/

Kafka is generally used for two broad classes of applications: *(i)* building real-time streaming data pipelines that reliably get data between systems or applications; *(ii)* building real-time streaming applications that transform or react to the streams of data [7].

The ML algorithm micro-services, as mentioned before, must be wrapped so that they include an internal connector to obtain the training sets supporting multifarious sources as well as an external connector that sends the trained model, once the job is finished, to the MUSKETEER server.

More in detail, in order to make client connector able to use data which is stored in different storages, it has to be made available a set of internal connectors to user's data sets, like:

- HIVE connector, to read data from or write data to Hive data sources. The Apache Hive data warehouse software facilitates reading, writing, and managing large datasets residing in distributed storage using SQL. Structure can be projected onto data already in storage [8].
- Cassandra connector to read data from or write data to Cassandra data sources, and enable the ingestion of temporal data in real time and maintain these records with a long retention period. The Apache Cassandra database is the right choice when scalability and high availability are needed without compromising performance. Linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure make it suitable for mission-critical data. Cassandra's support for replicating across multiple datacenters is best-in-class, providing lower latency for your users and the peace of mind of knowing that you can survive regional outages [9].
- Redis to read data from or write data to REDIS data sources. Redis is an open source, in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams [10].
- HDFS to read data from or write data to HDFS systems. It is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suit-

[7] https://kafka.apache.org/

[8] https://hive.apache.org/

[9] http://cassandra.apache.org/

[10] https://redis.io/

able for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is now an Apache Hadoop subproject [11].

Each ML algorithm will also send status update messages to the server using the MUSKETEER-Client python libraries.

[11] https://hadoop.apache.org/hdfs/

6 Conclusion

The purpose of this deliverable entitled *D7.1 - Client connectors' architecture design – Initial version*, was to deliver the design specifications of the client connectors in MUSKETEER. The deliverable is built directly on top of the first version of list of technical requirements presented in D2.1 and the knowledge extracted from deliverable D3.1 on the MSUKETEER general architecture, in order to deliver the details of the design of the client side of the integrated MUSKETEER platform.

The client connector architecture is micro-service oriented and it is designed in the form of cluster (dedicated virtual machines that are spawned on demand) so that each user is able to perform tasks in an isolated and secure environment. The client connector contains a set of interconnected components that constitute user's execution environment of the MUSKETEER platform.

The current deliverable presented the design specifications of the MUSKETEER client connector that were formulated following the International Data Spaces Association's Reference Architecture Model.

The alignment with the Industrial Data Platform standards brought forward by the Industry Data Space (IDS) Association guarantees that the MUSKETEER project outcomes will be interoperable with any other asset building on the IDSA standards.

However, as the project development activities evolve, this initial design of the described services composing the client connectors will receive the necessary updates and optimisations in order to encapsulate all the project's advancements, as well as the new technical requirements that will be extracted from the feedback that will be collected from the platform's evaluation.

Hence, the forthcoming versions of this deliverable will incorporate all the updates that are necessary to be introduced.

7 References

[1] Reference Architecture Model. Version 3.0. april 2019. IDSA. https://www.internationaldataspaces.org/wp-content/uploads/2019/03/IDS-Reference-Architecture-Model-3.0.pdf

[2] S. Newman (2015). Building Microservices – Designing Fined-Grained Systems, O' Reilly.