

**fortiss - Research Institute of the free State of
Bavaria for software intensive systems and services**



THINGS TO SERVICE MATCHING - TSMATCH

Technical Report

Authors: Erkan Karabulut (karabulut@fortiss.org)

Prof. Dr. Rute Sofia (sofia@fortiss.org)

IIoT competence field, fortiss GmbH, Munich, Germany
September 2022

Acronyms

IoT	Internet of Things	11
OneDM	One Data Model	10
TD	Thing Description	9
OGC	Open Geospatial Consortium	10
NLP	Natural Language Processing	6
WoT TD	Web of Things Thing Description	10
OneDM	One Data Model	10
SDF	Semantic Definition Format	10
NLTK	Natural Language Toolkit	9

List of Figures

2.1	TSMATCH software architecture.	7
2.2	Data Pre-processing Steps.	10
	11figure.caption.7	
2.4	An example service request that includes elements from FIESTA-IoT ontology.	12
2.5	Capturing and processing of a service request.	12
2.6	Ontology import sequence diagram.	13
2.7	Thing discovery and matchmaking sequence.	14
3.1	TurtleBot3 Burger robot in the fortiss IIoT Lab as the mobile IoT Thing.	17
3.2	Robot setup.	18
3.3	Home screen of the TSMATCH Android app.	25
3.4	Discover screen of the TSMATCH Android app.	25
3.5	Service selection screen of the TSMATCH Android app.	26
3.6	Service response screen of the TSMATCH Android app.	27
3.7	Monitoring screen of the TSMATCH Android app.	27
4.1	Demonstrator setup in the fortiss IIoT Lab.	30
C.1	Static IP addresses and login credentials of the TSMATCH hardware.	39

Contents

1 Introduction	6
2 TSMatch Software Architecture	7
2.1 TSMatch Engine	9
2.1.1 Semantic Matchmaking	9
2.1.2 Service Request and Data Aggregation	11
2.1.3 External Service Interface and Ontology Import	12
2.2 Things Discovery	13
3 How to Run TSMatch	15
3.1 Database	15
3.2 MQTT Broker	16
3.3 Examples of IoT Things	16
3.3.1 RPI Thing	16
3.3.2 Robot: Mobile IoT Thing	17
3.4 Thing Discovery	18
3.5 Semantic Matchmaking	19
3.6 Data Aggregation	20
3.7 Ontology Interface	21
3.8 Service Registry	22
3.9 RabbmitMQ Connector	22
3.10TSMatch Connector Node.js Library	23
3.11TSMatch Client	24
4 The fortiss IIoT TSMatch demonstrator	29
4.1 Hardware Setup	29
4.2 Communication	29
4.2.1 MQTT Topics	30
5 Summary	32
References	33
A TD to Ontology Element Matching Algorithm	34
B Source Code Documentation	36
C Static IP List	39

Executive Summary

This report covers the architectural design of the fortiss middleware TSMatch v2.0. TSMatch is being developed by fortiss as a tool that explores the application of semantic matchmaking to improve the data usage across IoT infrastructures.

TSMatch is based on a client-server architecture, where a user can rely on an application (currently available for Android) to obtain results from an IoT service of an existing platform. The server-side of TSMatch relies on Machine Learning and semantic matchmaking between sensor descriptions and ontologies, to provide a finer-grained measurement result to a specific IoT service.

The proposed solution is based on the assumptions that each IoT device has a semantic description, and that IoT services are also semantically described.

The report covers the design of the TSMatch v2.0 open-source software, which is available via https://git.fortiss.org/iiot_external/tsmatch.

The report is organized as follows. Chapter 1 briefly explains TSMatch, providing additional scientific pointers. Chapter 2 describes the TSMatch software architecture and components, explaining the interaction among the different components. Chapter 3 explains how to run the TSMatch software stack, while Chapter 4 goes over the set up of our demonstrator, which can be helpful to assist others in local replication. Chapter 5 concludes the report.

1

Introduction

Interoperability remains to be one of the main challenges in the IoT domain. The ever-increasing number of IoT devices from various vendors requires a consensus mechanism for different aspects. One of them being the semantic interoperability which refers to reaching a consensus on the IoT data such as sensor measurement aspects or measurement units.

TSMatch [1] is aiming to solve this semantic interoperability issue by providing an automated matchmaking algorithm between IoT devices and IoT services. The proposed solution is based on the following 2 assumptions; i) each IoT device has a semantic description, ii) each IoT service can be described semantically based on an ontology.

The previous version of the TSMatch was using Sørensen–dice coefficient and term frequency–inverse document frequency [2] [3]. However, this approach isn't capable of capturing the semantic meaning in a semantic description file. Therefore, it has been decided to make use of an Natural Language Processing (NLP)-based approach, which provides a better semantic matchmaking accuracy [4] [5].

2

TSMatch Software Architecture

An overall perspective on TSMatch, its components, interfaces, and connectors is provided in Figure 2.1.

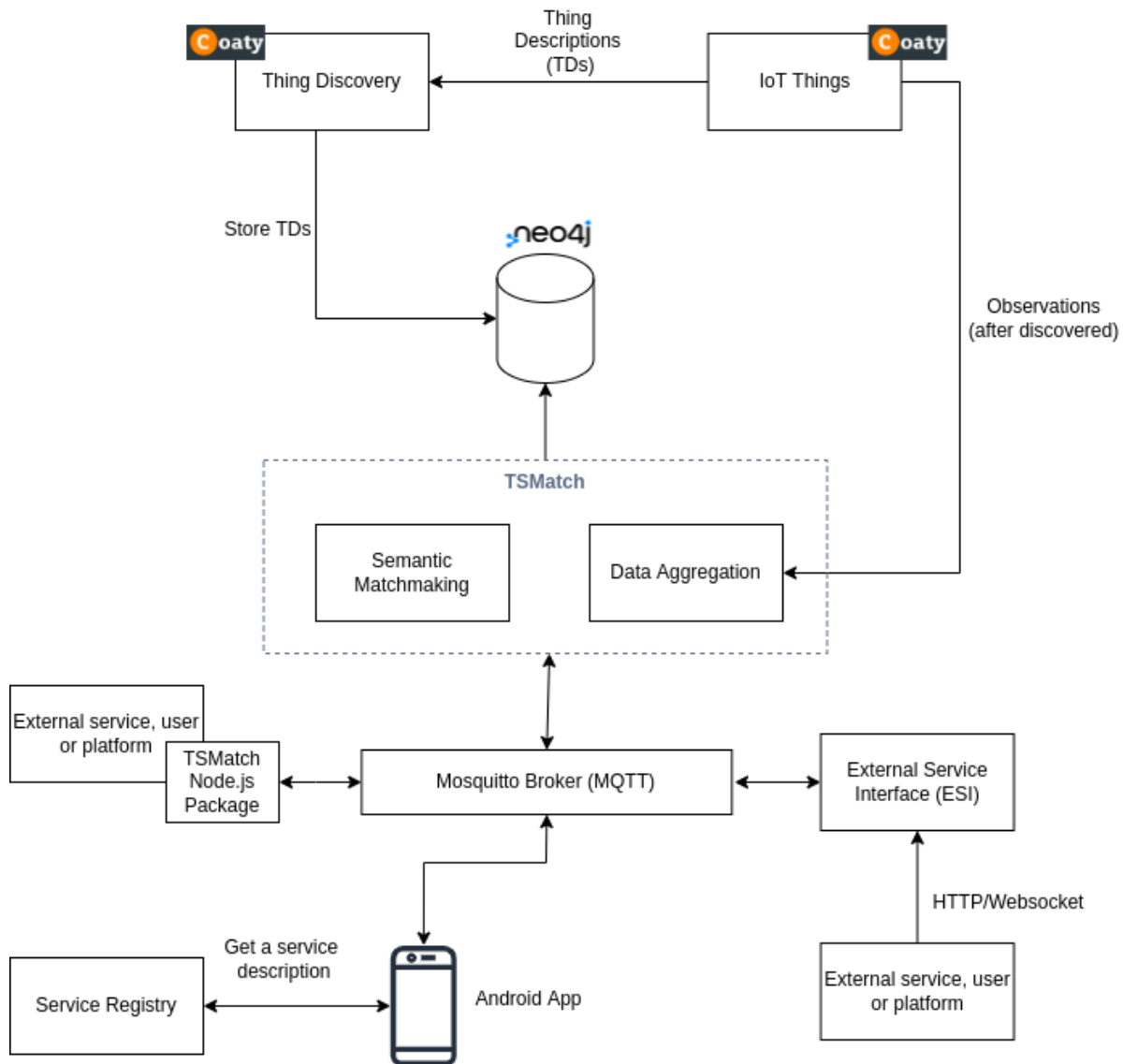


Figure 2.1: TSMatch software architecture.

The main goal of TSMatch is to automate the data exchange between IoT data sources and services, while satisfying the service needs.

Following a client-server approach, TSMatch comprises a server-side, the TSMatch engine, and a TSMatch client.

The TSMatch Engine (**tsmatch_engine**) is composed of 2 main functional blocks and several interfaces:

- **Semantic Matchmaking semantic_matchmaking**: Performs semantic matchmaking between IoT Things descriptions (stored on a database) and Ontologies. The result is a set of enriched data nodes, which are also stored in a database.
- **Data aggregation**: Sensor data aggregator.
- **Ontology interface**: Provides support for ontologies to be imported into TSMatch.
- **Connectors**: Different connectors, e.g., Mosquitto to RabbitMQ; HTTP/REST, etc.
- **External Service Interface**: Interface based on OpenAPI to interconnect to an external service registry.

The input and output of the matchmaking and data aggregation processes are stored on a local Neo4J database (**graphdb**), storing Things descriptions, service descriptions, ontologies, and new data nodes (aggregated Things based on a category, e.g., temperature measurement).

The TSMatch client (**tsmatch_client**) provides an Android app (source and binary). Moreover, TSMatch relies on the following external components:

- MQTT broker (**broker**). TSMatch currently relies on an MQTT broker based on Mosquitto as message bus. The tsmatch client and engine interconnect to the MQTT broker.
- Thing discovery (**thing_discovery**): IoT Thing discovery is supported via coaty.io¹
- Service Registry (**service_registry**): Holds a set of service descriptions. Currently holds environment monitoring service specification examples based on OWL and WSDL, which the user can select via the TSMatch client.

Moreover, the purpose of demonstration, the TSMatch stack includes the following:

- IoT Things: IoT Thing implementations using Raspberry Pi devices and physical sensors.

¹[https://coaty.io/](https://coaty.io)

- **Datasets (**datasets**):** All the datasets used in TSMatch:
 - Ontologies. We provide a copy of the FIESTA-IoT project which can be used to play with TSMatch.
 - Testing, Testing Cleaned, Training, Training Cleaned, holding a collected set of IoT Things descriptions that have been used respectively as testing and training sets.
 - word2vec, holding the wording that has been used to train the ML Word2Vec algorithm TSMatch relies upon.

2.1 TSMatch Engine

2.1.1 Semantic Matchmaking

The semantic matchmaking module consists of a data pre-processing step and a matchmaking of TD nodes to ontology elements step. Figure 2.2 shows the sub-steps of the prior.

Data pre-processing consists of the following steps:

- **Tokenization:** Separate text into sentences.
- **Remove punctuations.**
- **Fix camelCase:** Some words in the TDs are written in camelCase² format. A word written in this format might not exist in the word2vec model word list. Therefore they need to be separated into multiple words, e.g., "camelCase" to "camel case".
- **Lowercasing.**
- **Remove stopwords:** Python Natural Language Toolkit (NLTK) Toolkit³ includes a list of stopwords in English. Remove those words from the TDs.
- **Lemmatization:** Find stem of the words using "WordNetLemmatizer" from the NLTK toolkit.
- **Remove Trivial Words:** Some words that appear commonly in TDs create a non-realistic similarity, i.e., the word "sensor". As an example, the CBOW model produces a higher similarity score for "temperature sensor" and "luminance sensor" pair than "temperature" and "luminance".

After running each of these, a "cleaned" TD file is obtained. As a second step, the TD file is passed to a matchmaking algorithm that is given in algorithm A.

To better illustrate the concept developed to match Thing Description (TD)s using semantic matchmaking algorithms, figure 2.3 shows a part of a TD where a sensor is defined.

²https://en.wikipedia.org/wiki/Camel_case

³<https://www.nltk.org/>

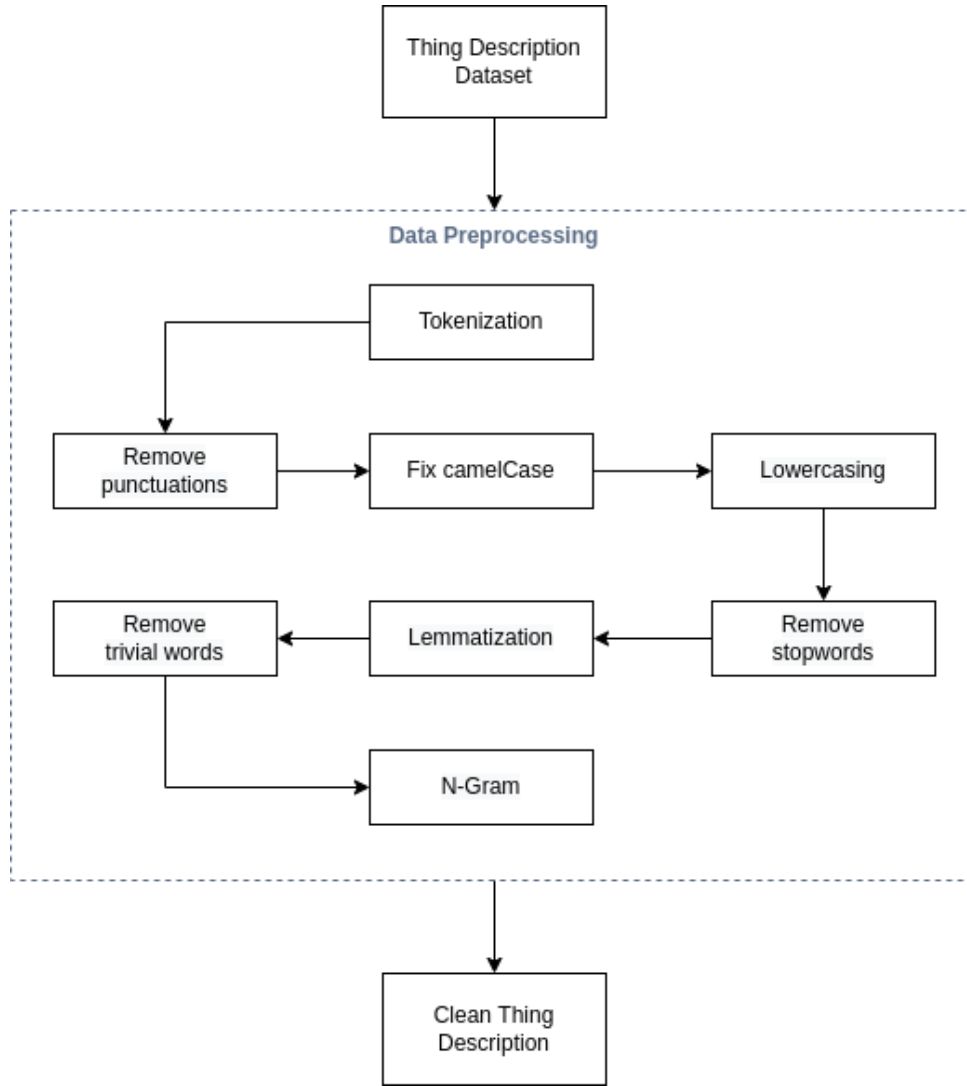


Figure 2.2: Data Pre-processing Steps.

Some attributes in a TD appear in different information models. Common terms are "name", "description" or "title" that exist in Web of Things Thing Description (WoT TD), Open Geospatial Consortium (OGC) SensorThings API Sensing part, and One Data Model (OneDM) Semantic Definition Format (SDF). These fields include valuable information regarding the type of the sensor, its measurement unit for instance. As an example, the sample TD shared in Figure 2.3 has name and description fields where it mentions the ontology "quantity kind" (Light intensity) and measurement unit (Lux) of a sensor. Therefore, the algorithm initially checks if a matching can be found using these fields in line 9. "ap["name"]" refers to the name of an aggregation point, e.g., "QuantityKind". "ap["category"]" refers to the children of an aggregation point, e.g., ["Temperature", "AirQuality", "Humidity", ...].

Each collected TD has one or more sensor description. Secondly, the algorithm checks if a matching can be found using these sensor descriptions in line 11. If this is also unsuccessful, then it checks the remaining parts of the TD to find a matching to the aggregation points that are extracted earlier in line 13.

Algorithm A describes how the semantic similarity approaches are used while

```

...
"sensor":{
  "name":"LightIntensity",
  "description":"Light intensity in Lux",
  "type":"object",
  "readOnly":true,
  "title":"LightIntensity",
  "properties":{
    "LightIntensity":{
      "type":"number",
      "readOnly":true
    }
  },
  "forms":[
    {
      "href":"https://w3ctest.iadstg.iot.ocs.oraclecloud.com/iot/api/v2/apps/0-AB/devices/-
113972C0-6CFC-4FBD-A586-87E7FFD385F1/deviceModels/urn%3Acom%3Aoracle%3Asolarpanel/attributes/-
LightIntensity",
      "contentType":"application/json"
    }
  ],
  "uuid":"51509897-9b2a-4b88-9b39-7f1cf19f8be1"
}
...

```

Figure 2.3: Part of a TD from the W3C WoT available testing sets ⁴.

matching a TD to child nodes of an aggregation point. The method named "similarity" refers to one of the 3 semantic similarity algorithms. First of all, the algorithm tries to find a match between keys in a TD and aggregation points between lines 4 and 10. For instance, it checks if there is a key that is similar to "QuantityKind" or "Unit". Each of the semantic similarity algorithm produces a similarity score between 0 and 1. In case the similarity score is higher than a KEY_SIMILARITY_THRESHOLD, then it is accepted as a matching key and consider the value of that KEY only in the next step.

In case there is an attribute that has a higher similarity to the given ontology aggregator point name, then the algorithm tries to find a matching value to the children of that aggregation point between lines 11 and 24. If there is no matching key then the algorithm checks similarity of each value in the given TD to the children of an aggregation point.

2.1.2 Service Request and Data Aggregation

Data aggregation is performed based on the assumption that every Internet of Things (IoT) service can be described semantically according to an ontology. This assumption suggests that a service request will include elements from an ontology. Upon receiving a service request, data aggregation module looks for matching sensing devices according to the ontological elements inside the request. Then the engine subscribes to data from those sensors and aggregate using a simple average function. This process is illustrated in figure 2.5 as a sequence diagram.

As an example, Figure 2.4 shows a sample service request which includes category elements from the FIESTA-IoT ontology. Data aggregation module searches for sensor descriptions in the graphdb that has a relation to "Illuminance", "Lux", "LightSensor" and "Environment" nodes. It sends a response back to the requestor that contains a list of sensor and TDs. Lastly, the module subscribes to data from those IoT devices, aggregates data using an average function (to be improved) and

```

{
  "QuantityKind": "Illuminance",
  "Unit": "Lux",
  "SensingDevice": "LightSensor",
  "DomainOfInterest": "Environment",
  ...
}

```

Figure 2.4: An example service request that includes elements from FIESTA-IoT ontology.

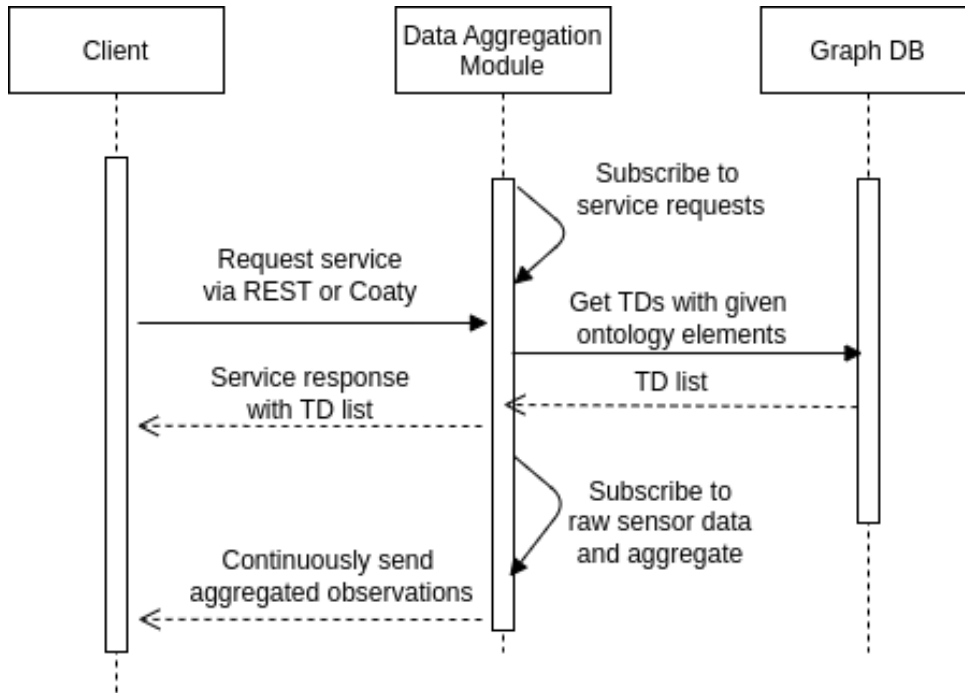


Figure 2.5: Capturing and processing of a service request.

continuously sends the aggregated data to the requestor until the service request is deleted.

2.1.3 External Service Interface and Ontology Import

The *External Service Interface (ESI)* module has been developed to support interconnection to external Service or Ontology Registries. For information on ESI, refer to TSMATCH ESI report.

For a simplified integration of ontologies, TSMATCH relies on the **ontology_interface**, which accepts the ontology data as a JSON object and also a Web address that hosts the JSON object.

For specific ontologies that we have considered, refer to the section on Datasets.

Figure 2.6 shows how a new ontology import process works. An external ontology data provider sends the ontology data to the `ontology_interface` module. The `ontology_interface` module then deletes the existing ontology and all of the matchings between ontology elements and the TD nodes.

It creates new nodes in the graph database per ontology element. Lastly, it

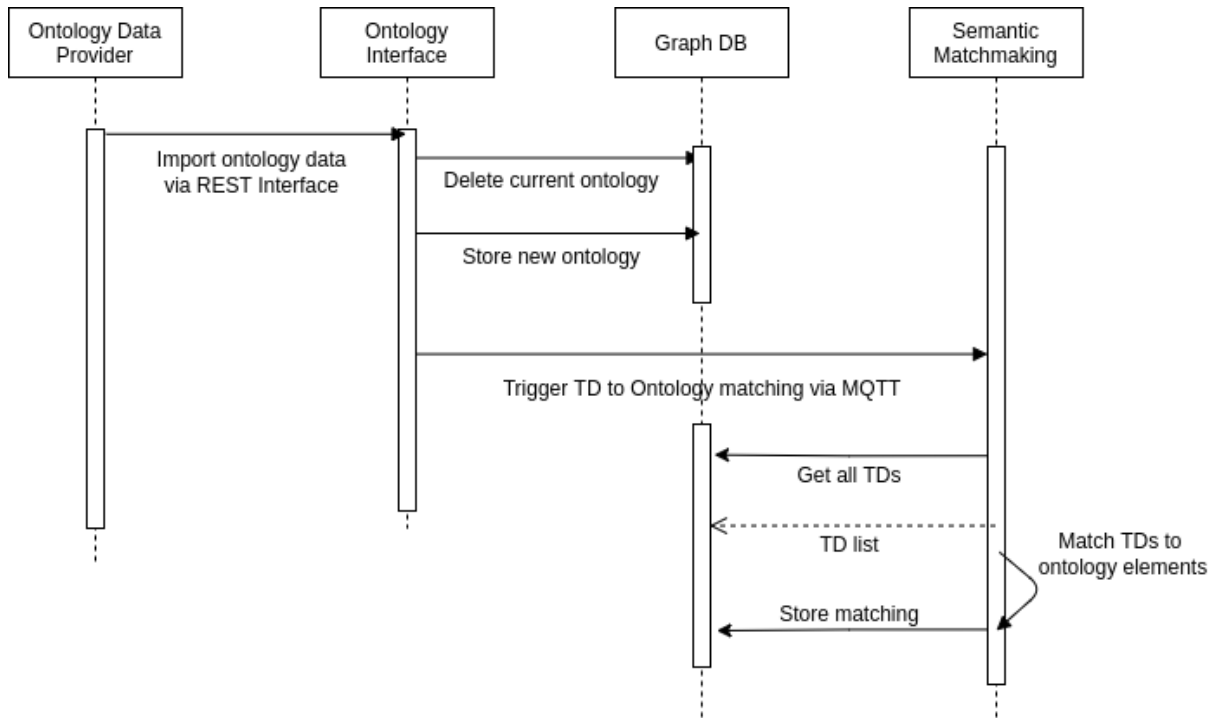


Figure 2.6: Ontology import sequence diagram.

informs the semantic matchmaking module regarding the ontology change. Semantic matchmaking module then re-runs the matchmaking process and matches TDs to new ontology nodes by creating relations in the graphdb.

2.2 Things Discovery

TSMATCH relies on external software to detect IoT devices in an infrastructure. This is done on a local scope (Local Area Network), via the framework Coaty.

Each IoT Thing (e.g., a Raspberry Pi with several environmental sensors) is represented by a standardized *Things Description (TD)*, and has a coaty agent installed. The TD is published over Coaty as part of a discovery event, when an IoT device boots up.

TSMATCH subscribes, via its **things_discovery** component (`../things_discovery`) for Coaty discovery events. Upon receiving a TD file, TSMATCH creates a new node in its GraphDB database (graphdb). Then the discovery module publishes a message over the MQTT broker on the "**fortiss-org.TSMATCH.NDATA.ONTOLOGY_CHANGED**" topic to notify the semantic matchmaking module. The matchmaking then matches the newly discovered TD to ontology elements in the database. Figure 2.7 illustrates the process of discovery and matchmaking.

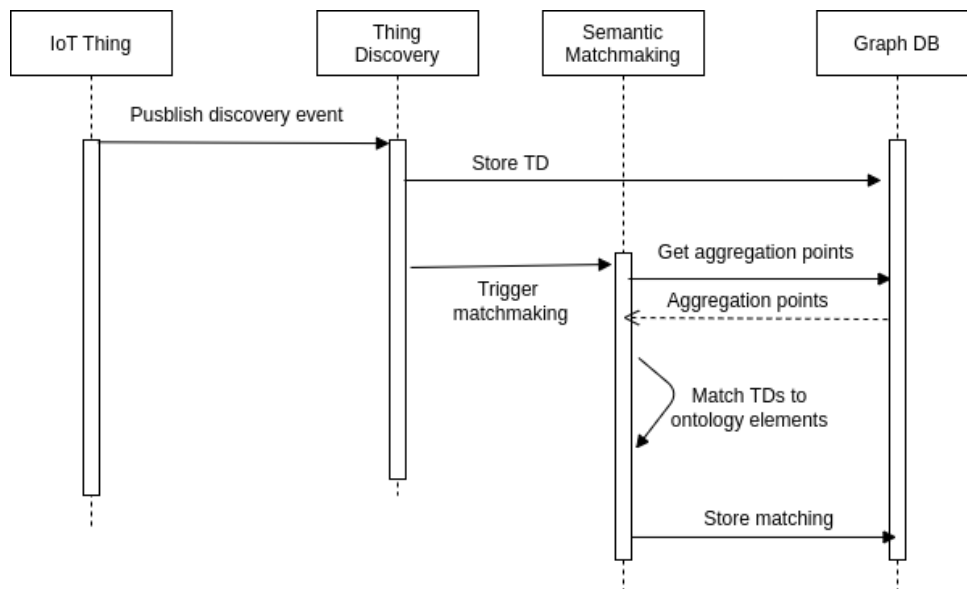


Figure 2.7: Thing discovery and matchmaking sequence.

3

How to Run TSMatch

This section explains how to run the TSMatch software stack.

The TSMatch stack can be run (on a single device) in an automatic way by starting the script "start_tsmatch_locally.sh" and "stop_tsmatch.sh". A list of modules run and stopped by these scripts are given below:

- Mosquitto Broker
- Neo4j Graph database
- Thing Discovery
- Semantic Matchmaking
- Data Aggregation
- ESI
- Service Registry

However, for the case of setting up the different components of TSMatch across different physical, resource constrained nodes, the containerized components can also be individually started. For this purpose, we have provided a runner and stop scripts for each of the modules separately as well. Each of these scripts are located in the main folder of respective folders. As an example, runner scripts for the Thing Discovery module is placed in "./thing_discovery" folder with the names start_thing_discovery.sh and stop_thing_discovery.sh.

3.1 Database

Neo4j¹ has been selected as the graph database to be used as a result of a prior analysis of existing graphDB databases. The Official Docker image of Neo4j13 is used to implement the database component (database). A default username-password pair can be provided in "./graphdb/start_graphdb.sh" by changing the line 13

"--env NEO4J_AUTH=neo4j/pass". In this line "neo4j" and "pass" correspond to username and password respectively.

¹https://hub.docker.com/_/neo4j

The database uses the default “7687” port to communicate. There is also a graphical user interface which allows to better interact with the database and also visualize the data. The GUI is published on “7474” port. Run the following commands to start the database container:

```
cd graphdb
chmod +x start_graphdb.sh
./start_graphdb.sh
```

3.2 MQTT Broker

We rely on Mosquitto² to implement a message broker that provides the messaging infrastructure for the rest of the system. It requires a username and a password to connect to the broker. A default user is created on initialization of the broker. The broker is implemented inside a docker Alpine image and the default username password is passed as an environment variable inside the “./broker/.env” file. Environment variables must be set before running the broker.

There is a configuration file located at “./broker/mosquitto.conf” which specifies the default user and allows web socket communication over the port “9001”. The default port for MQTT communication is “1883”. Web socket is used to communicate with the mobile app. The broker can be run with the following command:

```
cd broker
chmod +x start_broker.sh
./start_broker.sh
```

3.3 Examples of IoT Things

3.3.1 RPI Thing

For the purpose of supporting the use of TSMatch, we provide 2 Things implementations. The first is based on a Raspberry Pi (RPI) device with different sensors attached to it, sensor driver software which reads data from sensors, and a coaty.io agent which publishes sensor data over the broker. Rf. to chapter 4 on the current setting of our IoT Things in the fortiss Labs.

The Thing component has been developed using Nodejs/Typescript and it is also containerized using an Alpine docker image as a base image. TDs are written inside “src/controller/sensor—things—controller.ts”, and sensor descriptions are written inside “src/sensor/<sensor_name>.ts” files. It receives the broker connection info as environment variables inside the “./thing/.env” folder. All of the environment variables must be set before running the software. Follow the commands below to run the stationary IoT things in a RPI device. The Thing provides synthetic temperature data, until 20C - this can be changed in “src/controller/sensor—things—controller.ts”.

²<https://mosquitto.org/>

To run it inside a docker container (recommended):

```
cd thing
chmod +x start_thing.sh
./start_thing.sh
```

To stop the docker container:

```
chmod +x stop_thing.sh
./stop_thing.sh
```

To run it using NPM:

```
cd thing
npm install
npm run build
npm start
```

3.3.2 Robot: Mobile IoT Thing

A TurtleBot3 Burger robot³ (see figure 3.1) is used to create a mobile IoT Thing. The robot consists of a Single Board Computer (Raspberry Pi), OpenCR robot controller, 2 DYNAMIXEL motors for wheels, 2 wheels, Li-Po battery, 360 degrees LiDAR and some physical, scalable structures to build the robot.

The robot uses the ROS Noetic⁴ operating system. Roscore⁵ which is a collection of nodes and programs that are prerequisites of a ROS-based system is installed on the NUC device where the database and the broker is installed. And a node⁶ is installed on the RPI that is attached to the robot to perform computations and also to control the robot. This setup is shown in Figure 3.2.

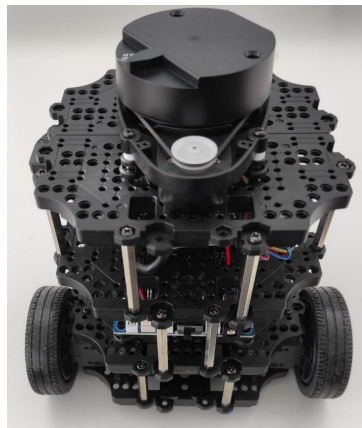


Figure 3.1: TurtleBot3 Burger robot in the fortiss IIoT Lab as the mobile IoT Thing.

A Coaty agent and a virtual temperature sensor is installed on the RPI. In order to run the robot, first place it in the middle of the Decentralized Edge Data

³<https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/>

⁴<http://wiki.ros.org/noetic>

⁵<http://wiki.ros.org/roscore>

⁶<http://wiki.ros.org/nodes>

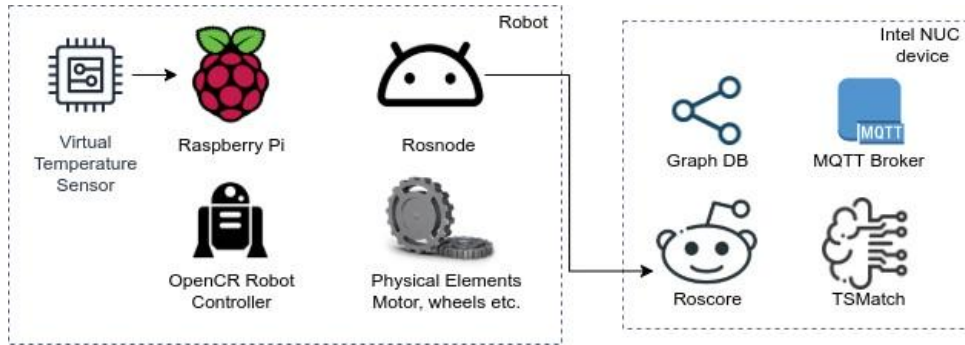


Figure 3.2: Robot setup.

Exchange Demonstrator stand. Second, run the following command on the NUC device to boot the roscore:

```
roscore
```

Thirdly, run the following command on the RPI to boot the robot:

```
roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

Currently in this initial setup of the Mobile IoT Thing, there are 2 ways to move the robot. i) Moving the robot around the table using a rule-based code, ii) moving the robot to the right and left sides of the table continuously with a coordinate-based code.

For the first method, place the robot on the upper-right side of the stand and run the following command on the RPI:

```
python3 ./robot/robot_movement/src/rotate.py
```

For the second scenario, place the robot on the middle of the stand and run the following command on the RPI:

```
python3 ./robot/robot_movement/src/send_goal.py
```

3.4 Thing Discovery

The main goal of the thing_discovery component is to assist in automatically detecting local IoT Things.

The module contains a coaty.io agent that is placed under "src/service/Agent.ts". There are 2 Coaty controllers; one for capturing the TD itself (src/service/SensorController.ts and another one for capturing location objects (src/service/FeatureOfController.ts. It is written using Repository pattern and all of the classes that interact with the db is placed under src/repository folder.

Before running the Thing discovery module, the environment variables must be set inside .env file. The module requires MQTT broker connection parameters, topic names and the graph db connection parameters. After setting the parameters, the module can be run using the following scripts.

To run in it inside a docker container (recommended):

```
cd thing_discovery
chmod +x start_thing_discovery.sh
./start_thing_discovery.sh
```

To stop the docker container:

```
chmod +x stop_thing_discovery.sh
./stop_thing_discovery.sh
```

To run it using NPM:

```
npm install
npm run build
npm run service
```

3.5 Semantic Matchmaking

The semantic matchmaking module listens for IoT Thing discovery messages published by the Thing Discovery module over the "fortiss—org.TSMATCH.NDATA.DISCOVERY" topic. Upon receiving a TD object, it matches the TD to the ontology nodes which were previously imported into the TSMatch.

In order to match the TDs to ontology elements, the module uses the algorithm given in A. The algorithm uses Word2Vec NLP model to infer semantic meaning between a TD file and an ontology node. A basic structure of the algorithm in steps given below:

1. Extract "name", "description", and "title" keys and their values from the TD. Compare them to the ontology node names by running the proposed Word2Vec approach.
2. If the Word2Vec approach produces a similarity score that is higher than a predefined threshold, then accept it as a match.
3. If no match is found in the first two steps, then repeat the same comparison this time on all of the sensor description object. (A TD contains metadata for sensors and actuators among others)
4. If not match is found when using the entire sensor description, then simply compare the entire TD object with each of the ontology node name.

The algorithm currently has two thresholds. A first threshold is used to provide a comparison at the key level of the TD object to the centroids of an ontology (e.g., categories). A second threshold is used to provide a comparison at the value level of the TD object towards the children of the centroids. The intuition behind is that when a key is found similar to a centroid node name, then only the value of this key is considered for matching. As an example, if there is a key called "Unit" and one of the centroid nodes in an ontology is "MeasurementUnit", then it is clear that the key "Unit" contains the measurement unit for this sensor. Therefore, it is not required

to look into the other keys. An evaluation of different key and value thresholds has been performed and is available via [./Documentation/MSc_thesis.pdf](#).

The source code for the Word2Vec algorithm is placed in "src/algorithm/Word2Vec.py". The data pre-processing steps that are mentioned in section 2.1.1 can be found in "src/preprocessing/StringPreprocessing.py". Similarly to the other modules, all of the database related operations are placed under the "src/repository" folder.

Before running the semantic matchmaking module, the environment variables must be set. The module requires Neo4j graph db connection parameters, MQTT broker connection parameters and topic names, and key-value similarity thresholds. After setting the parameters, the module can be run as described below:

To run it inside a docker container (recommended):

```
cd semantic_matchmaking
chmod +x ./start_semantic_matchmaking.sh
./start_semantic_matchmaking.sh
```

To stop the docker container:

```
chmod +x /stop_semantic_matchmaking.sh
./stop_semantic_matchmaking.sh
```

To run it using Python:

```
cd semantic_matchmaking
python3 src/main.py
```

3.6 Data Aggregation

The main objectives of the data aggregation module is to subscribe service requests, find matching Things Descriptions in the database, to then aggregate the raw sensor data, providing a measurement to the service request received from the TSMatch client. A service request can also be deleted by sending its UUID to a the delete service request topic. Figure 2.5 shows this process as a sequence diagram.

One of the assumptions that has been stated in Chapter 1 is that each service can be described semantically. Therefore, the service requests are expected to hold strings that can be matched to ontologies.

Upon receiving a service request, this component performs a query into the database and looks for the aggregated nodes that have been created as a result of the Semantic Matchmaking process (rf. to section 3.5). It then subscribes to raw data from the respective aggregated sensors, and provides an average value based on the location parameter given in the service requests.

The module has two event handler classes for observation-published and service-request-received events. These classes are "src/handler/ObservationEventHandler" and "src/handler/ServiceRequestEventHandler" respectively. Similar to the previously described modules, all of the graph db interactions are gathered under "src/repository" folder.

The connection parameters for database and MQTT broker access are again passed as environment variables, which can be set inside ".env" file in the main directory of the data aggregation module. After setting the parameters, the module can be run as follows:

To run it inside a docker container (recommended):

```
cd data_aggregation
chmod +x ./start_data_aggregation.sh
./start_data_aggregation.sh
```

To stop the docker container:

```
chmod +x ./stop_data_aggregation.sh
./stop_data_aggregation.sh
```

To run it using Python:

```
cd data_aggregation
python3 src/main.py
```

3.7 Ontology Interface

The ontology interface runs as part of the ESI module. The full ESI module is described on another report.

On TSMATCH v2.0, we focus on the ontology interface component (**ontology_interface**). The purpose of the ontology interface is to allow user import a new ontology data into TSMATCH.

The ESI module accepts connections on 3003 and 3004 ports, which can be changed inside the "Dockerfile".

To import ontologies to the database it is to consider a neo4j browser and rely on port 8080:

```
http://yourhostIP:8080/ontology/
```

On the browser, select the command PUT and upload the ontology in JSON format. Some examples for ontologies are provided in the ontology folder under *ontology_interface*.

The API expects either a "url" parameter or a "data" parameter which is a JSON object that includes the ontology data. The URL for the ontology data can either refer to an "OWL" file or to a "WSDL" file.

To run it inside a docker container (recommended):

```
cd esi
chmod +x ./start_esi.sh
./start_esi.sh
```

To stop the docker container:

```
chmod +x ./stop_esi.sh
./stop_esi.sh
```

To run it using Node:

```
cd esi
Node index.js
```

3.8 Service Registry

One way of creating a service request in TSMatch is to send an OWL or WSDL file which describes an IoT service, via the TSMatch client. This file is then parsed and converted to a JSON object.

The service registry keeps a set of service description files and serves them over HTTP. It is implemented using Express.js⁷. A REST API that serves over the root path `"/:file_name"` which accepts a file name as the path parameter. The module then searches for the given file among the registered files and returns the content of it in case it is found.

Currently, there are 2 service description files, one in WSDL and the other one in OWL format. The service descriptions can be found under the `"/files"` folder. In order to add a new service description into the registry, simply place the description file into this folder. If the registry is running inside a docker container, then the container should be re-created.

To run it inside a docker container (recommended):

```
cd esi
chmod +x ./start_service_registry.sh
./start_service_registry.sh
```

To stop the docker container:

```
chmod +x /stop_service_registry.sh
./stop_service_registry.sh
```

To run it using NPM:

```
cd service_registry
npm start
```

3.9 RabbmitMQ Connector

The coaty.io framework doesn't fully support AMQP (underlying protocol for RabbitMQ). TSMatch is being applied in projects, such as in the EFPP project⁸, where the message bus is based on AMQP (RabbitMQ). Therefore, a connector software is required to connect the Mosquitto MQTT broker and the RabbitMQ broker used in the EFPP platform.

It simply redirects requests from the EFPP platform to Mosquitto and redirects the responses and discovery events to EFPP platform. Because of that, it requires addresses and credentials for both of the brokers. They can be set inside

⁷<https://expressjs.com/>

⁸<https://www.efpf.org/>

the `./env` file.

To run it inside a docker container (recommended):

```
cd connectors/rabbitmq_connector
chmod +x ./start_rabbitmq_connector.sh
./start_rabbitmq_connector.sh
```

To stop the docker container:

```
chmod +x /stop_rabbitmq_connector.sh
./stop_rabbitmq_connector.sh
```

To run it using NPM:

```
cd connectors/rabbitmq_connector
npm start
```

3.10 TSMatch Connector Node.js Library

In order to enable TSMatch connectivity in code level, e.g. programming by using the TSMatch functionalities, a Node.js library is created. All of the functionalities of the TSMatch can be used via this library. A set of the APIs provided by this library is given below:

- `TSMatchConnector.subscribeThingDiscovery(callback)`: Subscribe to Thing discovery events. The given callback function is triggered when a new IoT Thing is discovered, together with a TD object.
- `TSMatchConnector.subscribeThingRemoval(callback)`: Subscribe to Thing deadvertisement events. Whenever an IoT Thing is disconnected, the given callback function is called together with its TD object.
- `TSMatchConnector.unsubscribeThingDiscovery()`: Unsubscribe from the Thing discovery events.
- `TSMatchConnector.unsubscribeThingRemoval()`: Unsubscribe from the Thing deadvertisement events.
- `TSMatchConnector.requestAllThings(callback)`: Request all available IoT Things. The given callback function is triggered with an array of available IoT Things.
- `TSMatchConnector.serviceRequest(requestDescription, responseCallback, observationCallback)`: Send a service request to the TSMatch (requestDescription). When a response is sent by the TSMatch, then the responseCallback function is triggered together with the response object. After started receiving observations, then the observationCallback function is triggered together with observation objects.

- `TSMatchConnector.deleteRequest(deleteRequest)`: Delete a given service request by its object ID.

A TSMatch connector object can be created as follows:

```
import TSMatchConnector from "tsmatch-connector";
let connectionInfo = {
  url: "mqtt://localhost:1883",
  options: {
    username: "admin",
    password: "password"
  }
}
let connector = new TSMatchConnector(connectionInfo,
  ((isConnected: boolean) => {
    if (isConnected) {
      // successfully connected to the given TSMatch instance
    }
  }));
```

The following code snippet is an example of how to use the provided APIs:

```
connector.subscribeThingDiscovery(((payload: any) => {
  console.log("A new IoT Thing is discovered: ",
    payload.toString())
}));
```

It subscribes to Thing discovery events by using the connector object that is generated earlier. And the payload object refers to a Thing Description object of a newly discovered IoT Thing.

3.11 TSMatch Client

The TSMatch client has been implemented in Android and tested on Android versions 4.4.1 and above.

The application integrates 4 different pages:

- Home page: Select an environment to connect to. Currently it provides automatic access to the fortiss IIoT Lab or EFPF platform.
- Discover page: List available sensors in the selected environment.
- Service selection page: Describe and send a service request to TSMatch, and see a list of existing service requests.
- Monitor: See graphical representations of the aggregated sensor data per service request and location.

The home screen, shown in figure 3.3, allows users to connect to either the TSMatch instance in the fortiss IIoT Lab, or to EFPF platform directly. Clicking on

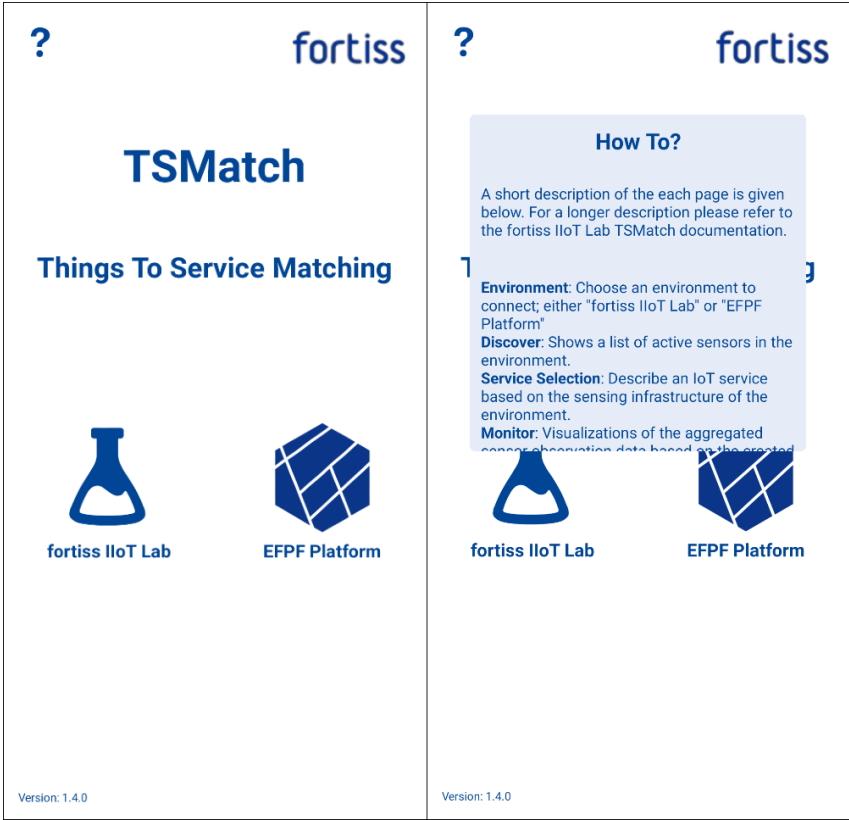


Figure 3.3: Home screen of the TSMatch Android app.



Figure 3.4: Discover screen of the TSMatch Android app.

the question mark icon at the top left of the home screen, opens up a model which explains what users can do with the app.

Figure 3.4 shows the discovery screen of the app. All of the local available sensors are listed on this page, together with their name, description and a logo showing the type of the sensor. As shown on the right hand side of the figure, clicking on a sensor opens up a modal that contains more detailed information; name of the IoT Thing, it's location and measurement unit.

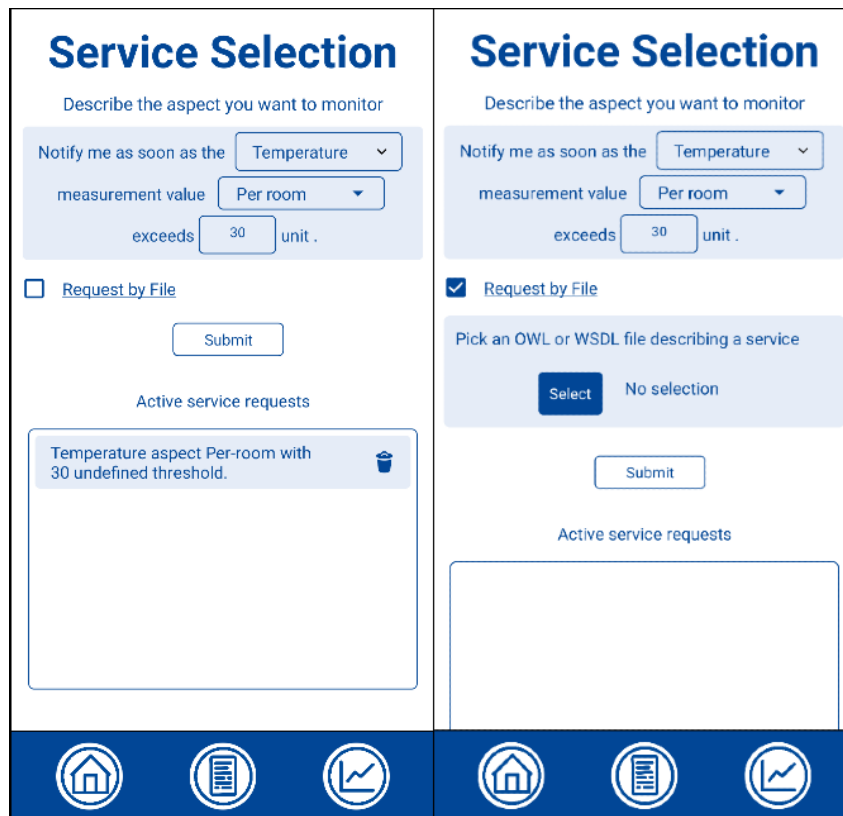


Figure 3.5: Service selection screen of the TSMATCH Android app.

Service selection screen, shown in figure 3.5, allows users to describe a service in the form of a sensor. Users can select a measurement aspect, location for data aggregation and a threshold value. If the sensor measurements exceed the given threshold, then the mobile app starts to send notifications. It is also possible to import an OWL or WSDL file describing a service request, as shown on the right hand side of the same figure.

After submitting a service request and receiving a response from TSMATCH, the response is shown in a separate window (see figure 3.6). This page shows that some measurement aspect (temperature in this case) successfully monitored, mean function is used for data aggregation and a list of sensor names from which the measurement data is collected. In case there are no sensors available to provide the requested service, then a message saying grouping isn't possible is shown on this page.

Lastly, the monitor screen shows the aggregated sensor data published by the data aggregation module. As shown in figure 3.7, the data is visualized as a line graph. On top of each graph, the monitored measurement aspect as well as the

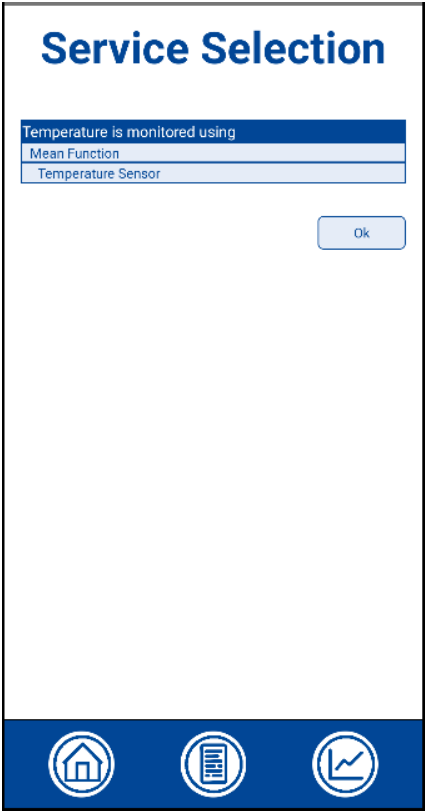


Figure 3.6: Service response screen of the TSMatch Android app.

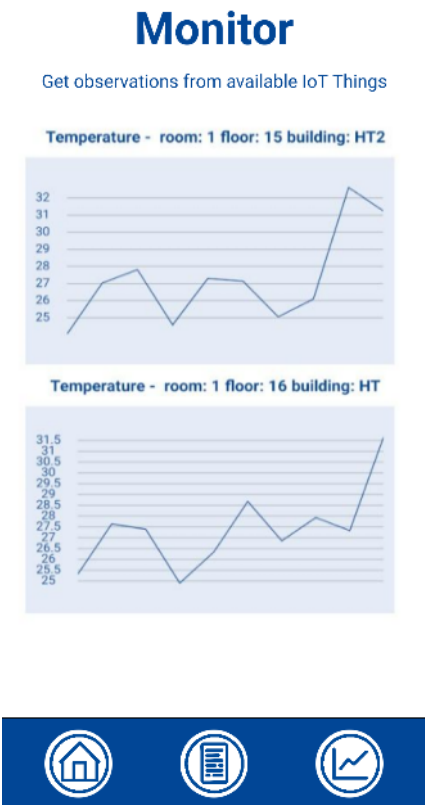


Figure 3.7: Monitoring screen of the TSMatch Android app.

location from which the data is collected is shown.

The application can be run on any Android device with version 4.1 and higher. An apk file is placed under the folder "tsmatch_client/android/app/build/outputs/apk/release". Requirements to be able run the application on an Android device is given below:

- Minimum SDK version of the app is 16. This means it supports Android 4.1 and above.
- In case there is no file manager in the smartphone, install one from any Android App Store. Some popular ones: EZ File Explorer, File Manager.
- Installing unknown apps must be allowed on the smartphone, since the app is not in an App Store. The way to allow unknown apps on a smartphone changes according to the Android version. How to do it is explained [here](#).

In order to install the app on an Android device that satisfies the above requirements:

- Connect your phone to a computer with a USB cable and move the APK file into your smartphone. Or download the APK file directly to your smartphone.
- With using a file manager, navigate into the folder where the APK file is located.
- Click on the APK file and follow the instructions of the APK installer on your smartphone.

4

The fortiss IIoT TSMatch demonstrator

This chapter provides information on the use of TSMatch on the fortiss IIoT Lab, including hardware, software aspects.

4.1 Hardware Setup

TSMatch consists of the following hardware components:

- Intel NUC device: An edge device that acts as the fortiss IoT gateway. In the current setup, it hosts the graph database and the MQTT broker.
- CSL Box: Another edge device that has lower resources compared to the Intel NUC device. It hosts TSMatch core components; thing discovery, semantic matchmaking, data aggregation, external service interface.
- Stationary IoT Things: There is 1 RPI 3B+ and 1 RPI 4B which have the following 5 sensors attached on them; low precision temperature and humidity sensor, high precision PT-100 temperature sensor, noise sensor, air quality sensor, and particulate matter sensor.
- Mobile IoT Thing: TurtleBot 3 Burger robot that has a RPI camera attached on it.
- Samsung smart phone: It runs the TSMatch mobile client.

A picture of the demonstrator in the fortiss IIoT Lab is shown in figure 4.1. Details of each hardware can be seen in this excel file.

4.2 Communication

This section provides an overview on how the software modules of the TSMatch communicate with each other. Details of the communication is given in the individual sections for each of the module. IP addresses and login credentials for each of the device listed above is given in Annex C.

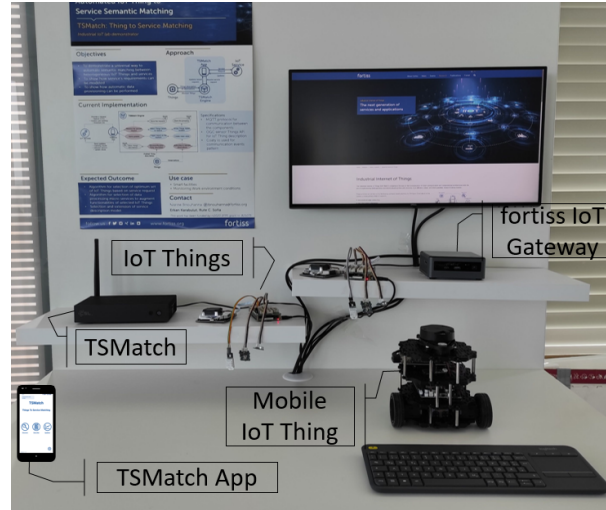


Figure 4.1: Demonstrator setup in the fortiss IIoT Lab.

Discovery of the IoT Things are handled by the Coaty IoT framework. When an IoT thing starts to run, it publishes a discovery event over Coaty, together with its TD. This event is then captured by the Thing Discovery module of the TSMATCH.

ESI is used to communicate with external services or ontologies, via HTTP + Websockets and MQTT protocols. This implies that in order to use the functionality of the TSMATCH, a client can either use HTTP+Websockets (please see this report for details) or MQTT protocol directly. Details of interacting with TSMATCH using MQTT protocol is given in sections below that are dedicated to each module of the TSMATCH.

Ontology interface inside the ESI is used to import new ontology data into TSMATCH. Upon receiving a request, and importing the new data, the ontology interface notifies the Semantic Matchmaking module over MQTT. It publishes a message to "fortiss-org.TSMATCH.NDATA.ONTOLOGY_CHANGED" topic which is then captured by the Semantic Matchmaking module.

The service requests to TSMATCH are captured by the Data Aggregation module. This module only accepts requests over MQTT. The ESI module interacts with the clients and publishes their request over MQTT in order for the Data Aggregation module to process it.

4.2.1 MQTT Topics

A list of all MQTT topics used for communication between TSMATCH modules and the clients are given below:

- Discovery - fortiss-org.TSMATCH.NDATA.DISCOVERY: TDs are published to this topic when a new IoT Thing is discovered.
- Deadvertise - fortiss-org.TSMATCH.NDATA.DEADVERTISE: When an IoT Thing leaves the network, it's TD is published into this topic.
- Service request - fortiss-org.TSMATCH.NDATA.SERVICE_REQUEST: Service requests to TSMATCH are published to this topic.

- Delete service request - fortiss-org.TSMATCH.NDATA.DELETE_SERVICE_REQUEST: In order to delete an active service request, publish a JSON object to this topic with "requestId" key containing the UUID of the service request.
- Response - fortiss-org.TSMATCH.NDATA.SERVICE_RESPONSE: This topic is for publishing the responses for service requests. The responses are published by the Data Aggregation module.
- Raw observation - fortiss-org.TSMATCH.NDATA.RAW_OBSERVATION: Raw observations from sensors are published to this topic.
- Aggregated observation - fortiss-org.TSMATCH.NDATA.OBSERVATION: After processing the service requests and raw observations, the data aggregation module publishes the aggregated observation values into this topic.
- Request all Things - fortiss-org.TSMATCH.NDATA.REQUEST_DISCOVERY_ALL: In order to get a list of all available Things in the system, publish any message to this topic.
- Send all Things - fortiss-org.TSMATCH.NDATA.RESPONSE_DISCOVERY_ALL: After sending a request to get all available Things, a list of Things are published to this topic if any.
- Ontology change - fortiss-org.TSMATCH.NDATA.ONTOLOGY_CHANGED: When the IoT ontology data is changed, then the ontology interface module publishes a notification over this channel. It is then captured by the semantic matchmaking module. This module then re-matches available Things to new ontology elements.

5

Summary

Interoperability is one of the major topics in IoT. Ever-growing number of deployed IoT devices as well as number of vendors requires a consensus mechanism for various aspects including the semantics. This semantic interoperability aspect in IoT refers to using IoT data from several data sources together in order to provide a specific service.

TSMatch is a fortiss middleware (TRL6) that aims to improving the integration of IoT devices into data backbones, via the use of ML-based semantic matchmaking.

TSMatch is being applied in pilots in the context of the EFPPF project, and can be downloaded for further use.

The TSMatch stack is developed in a micro-service architecture where each service is containerized and isolated from each other. The TSMatch stack has been developed to be able to run on far Edge devices.

References

- [1] N. Bnouhanna, R. C. Sofia, and A. Pretschner, "IoT Thing to Service Semantic Matching," in *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. IEEE, 2021, pp. 418–419.
- [2] N. Bnouhanna, E. Karabulut, R. C. Sofia, E. E. Seder, G. Scivoletto, and G. Insolubile, "An Evaluation of a Semantic Thing To Service Matching Approach in Industrial IoT Environments," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 2022, pp. 433–438.
- [3] N. Bnouhanna, R. C. Sofia, and E. Pristeri, "Applying MQTT Sparkplug in the EFPF Platform," *fortiss White paper*, 2021. [Online]. Available: https://www.researchgate.net/publication/358618553_White_Paper_Applying_MQTT_Sparkplug_in_the_EFPF_Platform
- [4] E. Karabulut, "ML-based Data Classification and Data Aggregation on the Edge," Master's thesis, Technical University of Munich (TUM) and fortiss GmbH, Munich, Germany, June 2022. [Online]. Available: https://www.researchgate.net/publication/361860458_ML-based_Data_Classification_and_Data_Aggregation_on_the_Edge
- [5] E. Karabulut, N. Bnouhanna, and R. C. Sofia, "ML-based data classification and data aggregation on the edge," in *in Proceedings of the CoNEXT Student Workshop (student poster)*, 2021, pp. 21–22.



TD to Ontology Element Matching Algorithm

Algorithm 1 TD to ontology element matching algorithm

```

1: procedure td_to_ontology_matching(thing_description)
2:   short_td = "name", "description", "title" fields in thing_description
3:   sensor_description = thing_description["sensor"]
4:   delete "sensor" in thing_description
5:   delete "name", "description", "title" fields in thing_description
6:   ap = get_aggregation_points()
7:   matching_dict = new Dictionary()
8:   for ap_name in ap do
9:     matching = match(short_td, ap["taxonomy"], ap["name"])
10:    if matching == None then
11:      matching = match(sensor_description, ap["taxonomy"], ap["name"])
12:      if matching == None then
13:        matching = match(thing_description, ap["taxonomy"], ap["name"])
14:      end if
15:    end if
16:    matching_dict[ap_name] = matching
17:  end for
18:  return matching_dict
19: end procedure

```

Algorithm 2 Find a match for a given thing description in a given category

```

1: procedure match(thing_description, category, ap_name)
2:   highest_score = 0
3:   matched_key = None
4:   for key in thing_description do
5:     score = similarity(ap_name, key)
6:     if score > highest_score then
7:       highest_score = score
8:       matched_key = key
9:     end if
10:  end for
11:  if highest_score > KEY_SIMILARITY_THRESHOLD then
12:    highest_score = 0
13:    matched_value = None
14:    for node_name in category do
15:      score = similarity(node_name, thing_description[matched_key])
16:      if score > highest_score then
17:        highest_score = score
18:        matched_value = thing_description[matched_key]
19:      end if
20:    end for
21:    if highest_score > VALUE_SIMILARITY_THRESHOLD then
22:      return matched_value
23:    end if
24:  end if
25:  highest_score = 0
26:  matched_value = None
27:  for key in thing_description do
28:    for node_name in category do
29:      score = similarity(node_name, thing_description[key])
30:      if score > highest_score then
31:        highest_score = score
32:        matched_value = thing_description[key]
33:      end if
34:    end for
35:  end for
36:  if highest_score > VALUE_SIMILARITY_THRESHOLD then
37:    return matched_value
38:  end if
39: end procedure

```

B

Source Code Documentation

This section includes file structure for the 3 TSMATCH modules that are developed in the current working period.

```

data_aggregation/                                // data aggregation module
  start_data_aggregation.sh                       // runs the module
  .env                                             // environment variables
  Dockerfile
  requirements.txt                               // required libraries
  src/
    main.py                                     // starting point
    repository/
      OntologyRepository.py                   // ontology related db operations
      ServiceRequestRepository.py            // service requests related db
                                              operations
      SensorRepository.py                   // sensors related db operations
      BaseRepository.py                     // common db operations, e.g.
                                              connect, disconnect

    service/
      ServiceRequest.py                     // active service requests
      MQTTClient.py                         // mqtt client
    handler/
      ObservationEventHandler.py             // handle an incoming observation
                                              event
      ServiceRequestEventHandler.py          // handle an incoming service
                                              request event

    util/
      Neo4jUtil.py                          // db related helper functions

ontology_interface/                              // ontology interface module
  manage.py                                    // django starting point
  .env                                         // environment variables
  Dockerfile
  start_ontology_interface.sh                 // runs the module
  requirements.txt
  ontology/                                    // sample ontology files
    m3-lite.owl
    ontology.json
    ontology.txt

```

```

    m3-lite.json
    converter/                                // owl to json converter
        owl2vowl.jar
        test.json
app/                                          // django app
    tests.py
    models.py
    admin.py
    apps.py
    views.py                                // views for each url mappings
    urls.py                                // url mappings
    repository/                             // db interactions
        OntologyRepository.py              // ontology related db operations
    service/
        MQTTClient.py                     // mqtt client
        OntologyService.py                // includes ontology import
                                           // operations
    util/
        StringUtil.py                     // string related helper functions
web/
    asgi.py                                // initialize django asgi app
    settings.py                            // django settings
    urls.py                                // url mappings
    wsgi.py                                // initialize django wsgi app

td_to_ontology_matching/                    // things description to ontology
                                           // matching module
    .env                                   // environment variables
    Dockerfile
    requirements.txt                        // required libraries
    start_data_enrichment.sh              // runs the module
    src/
        main.py                           // module starting point
        evaluate.py                       // performance evaluation
        algorithm/
            Clustering.py                  // k-means clustering
            Word2Vec.py                    // word2vec
            SentenceSimilarity.py          // lexical-db based sentence
                                           // similarity
        repository/                       // db interactions
            OntologyRepository.py          // ontology related db operations
            ThingRepository.py             // things related db operations
            SensorRepository.py            // sensor related db operations
            BaseRepository.py              // common db operations
        service/
            TDtoOntologyMatching.py        // match thing descriptions to
                                           // ontology elements
            MQTTClient.py                  // mqtt client
    preprocessing/

```

Word2vec.py	// word2vec related pre-processing operations
StringPreprocessing.py	// string pre-processing
util/	
StringUtil.py	// string helper functions
Neo4jUtil.py	// graph db helper functions
JSONUtil.py	// json helper functions
dataset/	
worst_final_clusters.json	// clusters using 1k word vectors
best_final_clusters.json	// clusters using 300k word vectors
testing/	// testing dataset
testing_cleaned/	// cleaned testing dataset
training/	// training dataset
word2vec/	// word vectors
google/	// word vectors by Google
training/	// trained word vectors using td training dataset



Static IP List

Table C.1: Static IP addresses and login credentials of the TSMATCH hardware.

IP Address	Device	Username	Password
10.0.33.39	Intel NUC	tsmatch	11otl4b4dm1n!?
10.0.33.41	RPI 4B	tsmatchthing1	11otl4b4dm1n!?
10.0.33.42	RPI 3B+	tsmatchthing2	11otl4b4dm1n!?
10.0.33.43	CSL Box	tsmatch	11otl4b4dm1n!?
10.0.33.47	Turtlebot 3 Burger	tsmatch	11otl4b4dm1n!?