FUTURISTIC BEEHIVES FOR A SMART METROPOLIS

Deliverable D1.2

Specification of external data & programming interfaces

Lead Beneficiary	LLU
Delivery date	24.03.2023
Dissemination Level	PU
Version	1.0
Project website	www.hiveopolis.eu

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 824069

DELIVERABLE SUMMARY SHEET

Project number	824069
Project Acronym	HIVEOPOLIS
Title	FUTURISTIC BEEHIVES FOR A SMART METROPOLIS
Deliverable No	D1.2
Due Date	Project month M48
Delivery Date	24.03.23
Name	Specification of external data & programming interfaces
Description	Communication interfaces and protocols for external system connectivity with a bio-hybrid living system will be specified. Several types of interfaces will be considered, like machine to machine and machine to human.
Lead Beneficiary	LLU
Partners contributed	BST
Dissemination Level	Public

Introduction

Purpose and scope of the document

The deliverable D1.2 presents an architecture for inter-HIVEOPOLIS unit communication and data exchange with external services hosted in the cloud. The aim of such a framework is to integrate into a common platform: HIVEOPOLIS units, external data sources, cloud hosted databases and models, user interfaces. The report provides an overview of the architecture, description of used protocols and specified data & programming interfaces, as well as example implementations of cloud services.

Overview of the document

This report starts by outlining key elements of the HIVEOPOLIS infrastructure. Chapter 2 focuses on protocol analysis for efficient communication with embedded devices. Chapter 3 specifies data & programming interfaces, describes examples of endpoints and package formats. Further chapters describe example implementations of cloud services and data & programming interfaces they provide.

Acronyms and Abbreviations

Acronyms and Abbreviations	**Definition**
ACL	Access Control List
CPU	Central Processing Unit
DSS	Decision Support System
GIS	Geographical Information System
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
LAD	Rural Support Service of Latvia
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
UUID	Universal Unique Identifier
WEB	World Wide Web
WFS	Web Feature Services
WMS	Web Map Service
XML	Extensible Markup Language

Infrastructure role in the HIVEOPOLIS ecosystem

The HIVEOPOLIS concept is not limited to developing highly sophisticated futuristic hives as stand alone units, but it also implies communication and data exchange between HIVEOPOLIS units, databases, user interfaces and external data sources.

The aim of task 1.4 is to provide an infrastructure for such communication and data exchange between various HIVEOPOLIS components via developing data and programming interfaces available for internal use and integrating required external data sources.

The main counterpart of the infrastructure is the Core module developed in WP3: it manages signal flows between hardware sensors and actuators, and locally hosts various software components (such as datastore and models). The latter need access to services for both external data acquisition and own data sharing and authorised publishing.

Other users of the infrastructure include a) archive and analytical databases hosted in cloud, b) augmented map service and models, c) various auxiliary services, such as notification or hardware monitoring utilities, d) integrated external data sources, such as weather forecast, e) external user interfaces and dashboards.

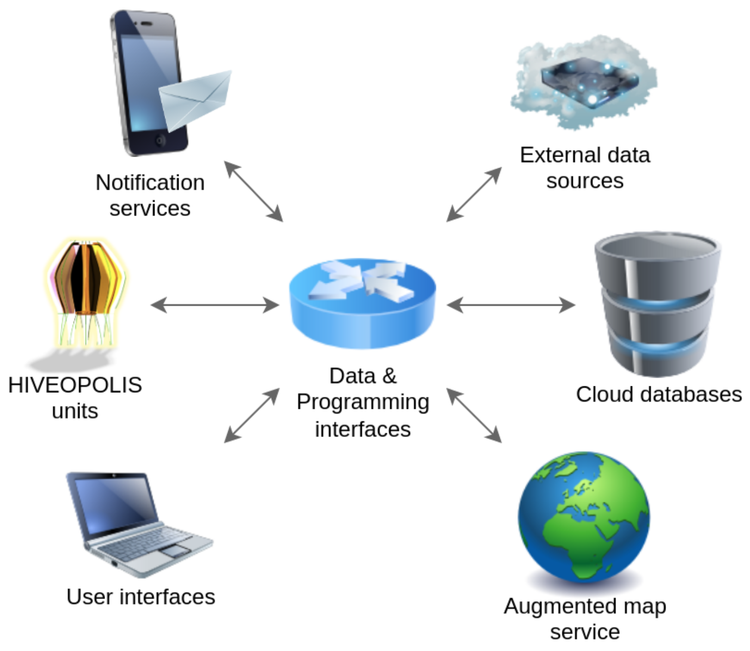


Figure 1: HIVEOPOLIS components interconnected via data & programming interfaces

This document provides specifications for data & programming interfaces, explains their implementation peculiarities and demonstrates application examples. The general communication architecture is specified, however particular data & programming interfaces cannot be considered complete nor comprehensive and are subject to change depending on further requirements and demands for data exchange.

Protocol analysis

Data exchange protocols define a set of rules for how various components communicate with each other. Specifications of data & programming interfaces depend on the type of technology used to implement them, thus the protocol framework should be selected beforehand. Various communication protocols are analysed depending on the level of application and the most suitable options are selected for implementation.

Constraints and limitations

A number of aspects were considered during analysis of protocol alternatives.

Embedded devices. The main clients for the data & programming interfaces are HIVEOPOLIS units (central core modules in particular). Core module (ref WP3) in its broad definition is an embedded device built upon Raspberry Pi single-board computer. Thus implementation of data & programming interfaces should consider limited computational resources.

Multiple platforms / programming languages. Besides HIVEOPOLIS units, other clients for data & programming interfaces will be various systems built for different platforms or architectures, for example, CPU architectures (ARM, x86, x64, etc), operating systems (Linux, Windows, Android, MacOS, etc). An additional level of diversity adds a variety of programming languages. Thus implementation of data & programming interfaces should be platform independent and support as many target platforms as possible.

Bidirectional data flow. By design data & programming interfaces should support data flow in both directions: to and from the client. Moreover, the initiator can be on any side of the communication channel.

Integration with the central core. Data & programming interfaces are designed in close relation with Data warehouse with implemented DSS (described in D3.3). Integration between these two data oriented systems was considered as a first priority feature.

Application layer protocols

In a simplified manner the task for data & programming interfaces can be described as transferring data packages from one network node to another. However, implementation of custom networking and/or transport protocol was not considered within the scope of the deliverable.

Also there could be good justification for implementing custom HIVEOPOLIS specific application layer protocol for data & programming interfaces, the task — data exchange between nodes — is quite a common use case for the majority of general purpose applications. Thus several available application layer protocols (and solutions) are analysed in this section.

HTTP. One of the widely used application layer protocols is Hypertext Transfer Protocol (HTTP). It is a text based request-response protocol for client-server systems widely used for WEB applications. All major platforms and architectures have various implementations both for client and server side components.

Despite widespread use of HTTP for WEB applications it has several drawbacks in a context of HIVEOPOLIS tasks. HTTP supports only peer-to-peer data exchange. However HIVEOPOLIS units (and other services) are considered as loosely coupled components with one-to-many communication strategy (broadcasting) being a common use-case. Thus a dedicated solution for message routing is needed.

The request-response nature of HTTP protocol implies that the communication initiator is a client, effectively limiting data exchange to one-way flow. However, there is a need for two-way data exchange among HIVEOPOLIS components (e.g. stream of sensory readings from hives to cloud services, or control commands from UI to hives). There are solutions for HTTP protocol that support two-way communication channels, such as polling or WebSockets.

Use of HTTP protocol one way or another implies the use of server software. Which depending on deployment configuration can be a limiting factor especially for embedded devices.

Message brokers are dedicated solutions for data exchange between various components. These brokers implement their own (often binary) communication protocols and support various message routing and processing schemas. Software packages such as ActiveMQ, Kafka, Redis can be mentioned as a few examples of message broker implementations.

In various sources message brokers are categorised as a special type of NoSQL databases and the key feature of these implementations is scalability. Message brokers are designed to support high-load systems. Also there is a potential need for high-load support in HIVEOPOLIS ecosystem in case it rapidly becomes widespread, but in a scope of project tasks such scenarios are not considered.

Similarly to the previous options, message brokers are software packages usually hosted on dedicated servers (or even clusters of servers) and are not suitable for embedded components.

MQTT is a lightweight protocol specially designed for IoT applications (Nastase, 2017). It has benefits of both previously mentioned types of protocols and solutions. MQTT implements message broker architectural pattern (publish-subscribe), but uses simplified networking schema, thus making it suitable for embedded devices with limited resources. MQTT design incorporates such use-cases as connectivity interruptions, optional message delivery assurance, flexible message topic hierarchy and others.

MQTT was evaluated in a context of integrating data & programming interfaces with the Core module. It is well suited for “local” inter-module communication within the Core (ref D3.3), and also supports communication with “remote” data & programming interfaces via bridging feature. Taking into account features of MQTT protocol, its relative simplicity, available implementations and expertise within the project it was selected as technology for application level protocol.

Similarly to other solutions MQTT operation relies on message broker software for routing the communication. Open source Eclipse Mosquitto message broker was used for this purpose. Its lightweight implementation is suitable for all types of devices and was deployed both on embedded Core modules and on the private hosting platform.

Payload format

Application layer protocols ensure message exchange between network nodes according to specific (application defined) rules. In case of MQTT these rules are publish-subscribe architecture pattern, message arrangement according to hierarchical topics, quality of service indicators, etc. However the content of the message itself — the payload — is not specified by the protocol and in case of MQTT is an array of bytes.

Few payload format options were considered for data & programming interfaces. One of the considered payload formats is **JSON** (*https://www.json.org/json-en.html*). It is a text based platform independent data exchange format widely used for Web applications. The main benefit of the JSON format is that messages are transmitted as human-readable text, thus simplifying debugging and problem investigation. The main drawback of the JSON is related to its strengths. As a text based format it is relatively verbose thus its use for message exchange will increase network traffic, which in turn is not desirable for embedded devices with limited connectivity like HIVEOPOLIS hives.

Also JSON is not ensuring any strict structure for the data messages. This fact makes JSON a very attractive option for prototyping phase while requiring additional efforts on message schema implementation and validation on later stages both on message sender and receiver sides.

With the aim to reduce message size binary message formats were considered such as **Protocol buffers**. Protobuf (Currier, 2022) is an open source data format for serialisation of structured data. Protobuf source code generation feature makes it available for a wide range of platforms and programming languages. At the same time low processing overhead promotes its use on embedded devices.

Flatbuffers is another binary serialisation format well suited for low memory implementations (Pradana et al., 2019) via “zero-copy” deserialization. While this feature makes data accessing faster and is especially important for embedded platforms, it requires more code for proper message handling. Also Flatbuffers are more restrictive on message contents because of fixed size field encoding. Taking into account the niche benefits and need for additional coding effort Flatbuffers were not further considered for implementation data & programming interfaces.

Payload formats were compared on relatively simple messages produced by embedded hive monitoring devices (more info in chapter Archive services). The devices on a regular basis are sending information about their battery charge, WiFi level, readings from load cell and multiple temperature sensors.

An example of a JSON encoded message is demonstrated below. Its minified version is **221 bytes** long.

```
{
  "battery": 3971.6294088924633,
  "modemBattery": 4102.063873291016,
  "scale": 73.0631794649012,
  "rssi": -79,
  "tempSensors": [
    {
      "serialNumber": 1025696878698943272,
      "value": 34.9375
    },
    {
      "serialNumber": 8591744252680639272,
      "value": 16.125
    }
  ]
}
```

Protobuf schema for encoding similar messages is as follows:

```
syntax = "proto3";

message TempSensor {
    float value = 1;
    uint64 serialNumber= 2;
}

message MeasurementNode {
  float battery = 1;
  float modemBattery = 2;
  float scale = 3;
  sint32 rssi = 4;
  repeated TempSensor tempSensors = 5;
}
```

The same payload serialised to binary string is **47 bytes** long. Its HEX representation is as follows:

```
0d83 3080 4515 5920 9242 189d 012a 0f0d
00c0 0b42 10a8 a6cf b087 bc80 9e0e 2a0f
0d00 0081 4110 a8a6 a2b0 87bc 809e 77
```

Code bindings for Python and C/C++ were generated from Protobuf schema. The latter were used for embedded software deployed on measurement devices.

Taking into account results of analysis Protobuf was selected as a payload format for both internal Core data exchange and communication with data & programming interfaces.

Message encryption

MQTT is a simplified communication protocol designed for embedded and IoT applications. However, as a flip side for simplicity, the security aspect of communication is not considered as a priority. As a result MQTT messages (including payload) are transmitted in plain text.

Insecure upstream of measurements from IoT devices might not have an impact on the overall system (apart from leaking private data together with measurements). However HIVEOPOLIS scenarios imply active control of units, thus making insecure messages a significant threat.

MQTT community suggests several ways to make the communication secure. The option widely used for IoT applications is the use of private networks, either physical or virtual via tunnelling. Effectively this option moves security implementation from MQTT to lower networking layers. However it needs dedicated infrastructure and centralised configuration of network nodes, which is not desirable for the HIVEOPOLIS ecosystem.

Another option is to keep MQTT messages insecure but use encryption (and validation) on a payload level. This provides a very flexible solution, but it requires additional efforts for implementing proper authentication and authorization mechanisms. Also this option ignores available authentication mechanisms available within the MQTT protocol itself.

The combination of mentioned options uses MQTT over TLS. It creates a secure transport channel between client and broker, and uses MQTT native authentication and authorization mechanisms. This approach requires no special configuration on the client side (except root certificates, which are available by default in all modern systems). Also the use of native MQTT authentication simplifies client side software. As a downside broker side requires a registered domain with a valid SSL certificate.

The latter option is used for implementing messaging between HIVEOPOLIS Core modules and components of data & programming interfaces. It was validated by analysing packages transmitted between client and broker using *tshark* utility.

```
tshark -i lo -f "dst port 8883" -d "tcp.port==8883,mqtt" \
    -Y "mqtt" -O "mqtt"
```

Insecure MQTT package was successfully decoded and plain text credentials obtained (marked in red).

```
MQ Telemetry Transport Protocol, Connect Command
    Header Flags: 0x10, Message Type: Connect Command
        0001 .... = Message Type: Connect Command (1)
        .... 0000 = Reserved: 0
    Msg Len: 22
    Protocol Name Length: 4
    Protocol Name: MQTT
    Version: MQTT v3.1.1 (4)
    Connect Flags: 0xc2, User Name Flag, Password Flag,
            QoS Level: At most once delivery (Fire and Forget),
            Clean Session Flag
        1... .... = User Name Flag: Set
        .1.. .... = Password Flag: Set
        ..0. .... = Will Retain: Not set
        ...0 0... = QoS Level: At most once delivery (0)
        .... .0.. = Will Flag: Not set
        .... ..1. = Clean Session Flag: Set
        .... ...0 = (Reserved): Not set
    Keep Alive: 60
    Client ID Length: 0
    Client ID:
    User Name Length: 5
```

```
        User Name: testB
        Password Length: 1
        Password: b
```

In case of MQTT over TLS only the header of the MQTT package is decoded.

```
MQ Telemetry Transport Protocol, Connect Command
    Header Flags: 0x16, Message Type: Connect Command
        0001 .... = Message Type: Connect Command (1)
        .... 0110 = Reserved: 6
    Msg Len: 3
    Protocol Name Length: 257
[Malformed Packet: MQTT]
```

The main drawback of MQTT over TLS approach is the traffic overhead needed for establishing TLS connection between client and broker. The encryption is implemented on the transport layer within TCP connection. Network nodes negotiate secret keys for encryption by interchanging with a number of technical messages, such as certificates and ciphers (so called handshake). During MQTT experiments low level network packages were monitored and it was found that the total overhead to establish a new TLS connection between client and broker is 6 567 bytes. In comparison, regular non-TLS connection initialization needs 668 bytes of technical messages. Further communication via encrypted connection doesn't have significant overhead and is comparable to regular TCP communication.

Architecture of data & programming interfaces

Overall the architecture of data & programming interfaces relies on MQTT protocol features and utilises MQTT broker implementation for message exchange between nodes. This chapter describes several MQTT specific features used for making the system extendable and simplify its maintenance.

MQTT broker can be considered as a central data exchange node which has primary responsibility to route messages between clients. In addition MQTT brokers are usually responsible for client authentication and authorization.

Within the HIVEOPOLIS project Mosquitto MQTT broker implementation is used, which also supports so-called bridging between several brokers. This feature allows building hierarchical snowflake-like data exchange structures. Lower level broker connects to another (higher level) broker as a regular client and routes messages from its own clients to a higher level and vice versa using a set of defined rules.

This feature is used to exchange messages between HIVEOPOLIS Core modules to Data & programming interfaces hosted in the cloud. Each Core module runs its own MQTT broker restricted only to local clients. This broker is not using any client authentication and can be used for internal purposes (ref D3.3). In addition, local brokers are configured to connect to the cloud broker using provided credentials and to map local topics with prefix `ho/#` to corresponding topics with prefix `ho/<username>/#`.

The cloud broker is a public service with disabled anonymous access. All clients connecting to the broker should pass authentication procedure according to a database of credentials and are authorised to publish/subscribe only to their own global subtopic. This ensures that clients are isolated from each other and private information is not leaking between clients. Also in case of faulty or malicious configuration of the client it won't be able to interfere with other subsystems due to topic restrictions.

In addition certain cloud services are authorised to access specific topics needed for their operation. For example, notification service is provided with read-only access to alert topics of all clients. In general such services are hosted in a controlled environment and have minimal required access rights for their operation.

An example of configuration for local broker is as follows:

```
listener 1883
allow_anonymous true

# bridge
connection hocloud
address science.itf.llu.lv:18883
remote_username hive1
remote_password password1
bridge_cafile /usr/share/ca-certificates/mozilla/ISRG_Root_X1.crt
topic # both 0 ho/ ho/hive1/
```

An example of configuration for cloud broker is as follows (in this case TLS connection is ensured by network infrastructure before the traffic reaches MQTT broker):

```
listener 1883
allow_anonymous false

# database of encrypted credentials
password_file /mosquitto/config/clients
# access control list
acl_file /mosquitto/config/acl
```

An example of access configuration (ACL) for cloud broker is as follows:

```
# notification service
user notification_service
topic read ho/+/alert/#

# read/write into own topics (%u is a placeholder for username)
pattern readwrite ho/%u/#
```

Figure 2 shows all mentioned aspects of data & programming interfaces, involved components and their interconnections. More details about specific hosted services are provided in further chapters.

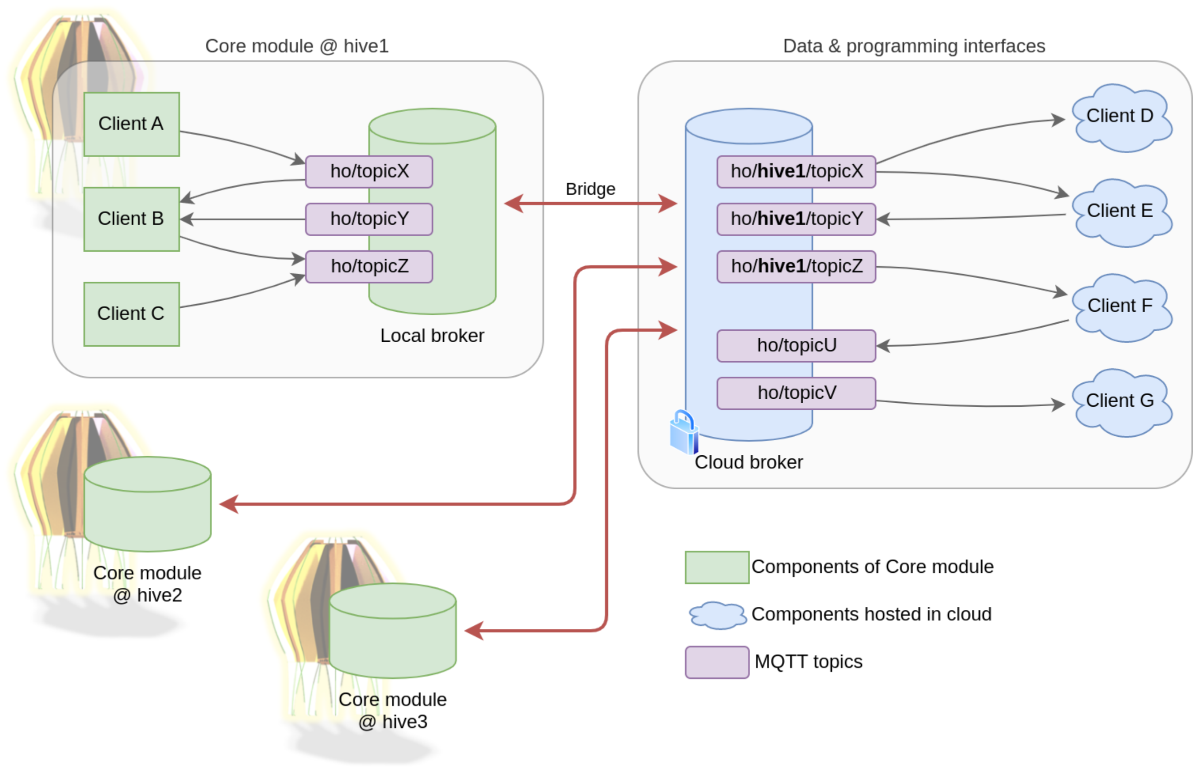


Figure 2: Overall architecture of Data & programming interfaces

MQTT topic hierarchy

MQTT protocol does not provide any naming conventions for topics. However for proper message delivery and processing, publishing and subscribing clients should agree on the same naming schema. In order to keep topic names organised across various HIVEOPOLIS components the hierarchy of topics is proposed. The hierarchy is not limited to mentioned cases and can be extended or modified according to specific requirements or use-cases.

The topics are described in the form of local names. If the topic is used in a global context, the client username is added to the prefix according to mapping rules where applicable. For example:

- **`ho/`**`live/sensors/broodnest/M49cRSbf`
  Local topic on hiveA
- **`ho/hiveA/`**`live/sensors/broodnest/M49cRSbf`
  Global topic available for external components

Live stream of messages

The idea behind these topics is to broadcast values from various sources immediately as they are obtained. Subscribers to these topics are able to process incoming data in a reactive manner or store them in a database for later use.

```
ho/live/sensors/:source/:id
```

Live stream of raw readings from various sensors. The source subsystem and sensor indicator (e.g. serial number or name) are provided for finer details about values. A few examples are as follows:

- `ho/live/sensors/broodnest/M49cRSbf`
  sensor readings from specific broodnest module (the comb with array of 64 temperature and 2 relative humidity sensors);
- `ho/live/sensors/entrance/0`
  number of bees passed through counting device at entrance (in/out);
- `ho/live/sensors/dancefloor/a1`
  dance detection report.

`ho/live/actuators/:source/:id`
Live stream of status reports from various actuators. Technically status reports are usually obtained from sensors built into actuators. However these readings are separated from the stand alone sensors to indicate "self-reported" origin of values. A few examples are as follows:

- `ho/live/actuators/broodnest/M49cRSbf`
  broodnest heating device actual and target temperatures, power consumption;
- `ho/live/actuators/gates/b9`
  status of specific gate (opening, closing, progress percentage).

`ho/live/system/:source/:id`
Live stream of various operating system parameters reported on a regular basis. A few examples are as follows:

- `ho/live/system/disk/raid0`
  report about used or free space on disk
- `ho/live/system/cpu/0`
  average load report
- `ho/live/system/cron/modelA`
  status of scheduled tasks (started, ended, failed, etc)

`ho/live/models/:id`
Modelling results reported immediately after computation, including model specific details, like hyperparameters, periods, forecast uncertainty, etc. A few examples are as follows:

- `ho/live/models/population`
  estimated dynamics of colony population
- `ho/live/models/state`
  detected colony state, like swarming, dead, queenless, active foraging, brood rearing, etc.
- `ho/live/models/storage`
  estimate of available honey for harvest

`ho/live/science/:experiment/:id`
Topics dedicated for status reporting of long-running scientific experiments. The need for such topics was identified for the prototyping and experimental phase of the HIVEOPOLIS project. A few examples are as follows:

- `ho/live/science/foraging-location/123`
  status updates on feeders located in various places
- `ho/live/science/wintering/987`
  status updates on comb heating experiment during winter period

Alert messages

The idea of these topics is to report important statuses to involved subscribers. The consumers of alert messages are notification services, user interfaces or automated safety mechanisms. The main difference between live messages and alerts is that the latter are reporting specific events or states that require human intervention.

`ho/alert/system/:source/:id`
Alert messages from various hardware and software components. A few examples are as follows:

- `ho/alert/system/disk/raid0`
  disk out of space
- `ho/alert/system/cron/taskA`
  scheduled tasks failed to start
- `ho/alert/system/broodnest/M49cRSbf`
  overheating detected on specific broodnest module
- `ho/alert/system/power/0`
  low battery level

`ho/alert/bees/:source`
Bee colony related alerts. These messages are distinguished from technology alerts and are related to biological processes of the bee colony and require beekeeper’s attention. A fFew examples are as follows:

- `ho/alert/bees/broodnest`
  cannibalism detected
- `ho/alert/bees/storage`
  combs are full of honey
- `ho/alert/bees/colony`
  population of colony decreasing rapidly

Data queries

Building a purely reactive system is not always desirable or even possible. Thus HIVEOPOLIS components need a mechanism for ad-hoc data queries. Within MQTT (and publish-subscribe architecture in general) such request-response interaction is usually implemented as a pair of related topics: one for request itself, and another for response or acknowledgement and status reporting.

`ho/query/:requestId`
`ho/query/:requestId/resp`
In order to distinguish multiple parallel queries from various sources, query topics are supplied with an unique request ID. There are no specific rules on how request ID is created, its sole purpose is to track response. Usually a random string is used as an identifier, like

UUID. This approach allows the request initiator to subscribe for responses only for its own queries.

There are possible multiple query engines — components responsible for processing incoming queries — which are fetching requested data from different data sources (e.g. from local database for recent data points and from archive database for longer history). Depending on the parameters different engines can handle particular queries, thus making data access simpler for requesting components (no need for details about internal organisation of data storages).

Augmented map service

Augmented map service can be considered as a special type of data source which provides ad-hoc access to results of spatial modelling instead of raw data points. Dedicated topics are used for different types of queries, but the overall approach is similar to regular query mechanism: pair of topics with unique request ID is used for data exchange.

```
ho/map/:queryType/:requestId
ho/map/:queryType/:requestId/resp
```

Few examples of queries are as follows:

- `ho/map/forecast/:requestId`
  weather forecast for requested GPS coordinates
- `ho/map/solar/:requestId`
  solar radiation parameters for requested GPS coordinates
- `ho/map/resources/:requestId`
  information about potential harvesting resources (flowering plants) for requested GPS coordinates

Command messages

Previously considered types of messages have informative nature and are used for data exchange between components. Logically distinct types of messages are dedicated for active commands for various actuators.

```
ho/cmd/:target/:id/:requestId
ho/cmd/:target/:id/:requestId/resp
```

Unique request ID is provided for tracking responses for specific commands. A few examples are as follows:

- `ho/cmd/gates/b9/:requestId`
  command: close gates
  `ho/cmd/gates/b9/:requestId/resp`
  response: ACK, in progress
- `ho/cmd/broodnest/M49cRSbf/:requestId`
  command: set heater target temperature to 42°C
  `ho/cmd/broodnest/M49cRSbf/:requestId/resp`
  response: NACK, not enough power

Additional types of messages and corresponding hierarchical topics can be added to the data & programming interfaces for specific use-cases.

Protobuf templates

Communication between components of data & programming interfaces relies on binary messages encoded according to Protobuf specification. However, the structure and content of the messages is defined by developers for specific use-cases. During the design and prototyping phase of data & programming interfaces several message structures were considered.

Protobuf message definitions allow so-called nested messages, when the message contains other messages as its fields (similar to classes in object oriented programming languages). Another feature available in Protobuf is Struct data type, which allows implementation of dynamically typed key-value maps.

Taking into account available features it is possible to build a universal definition suitable for any message type. Such a wrapper message has few header fields and a field for binary representation of any data object, including other Protobuf definitions.

```
import "google/protobuf/struct.proto";

message MessageWrapper {
    string module = 1;
    string instance = 2;
    int64 ts = 3;
    google.protobuf.Struct data = 4;
}
```

Despite being a very flexible and simple structure, such message design has a major flaw. The content and structure of the data field is not strictly specified thus requiring sending and receiving counterparties to agree on it outside of protocol specification. This effectively leads to implementation of custom data definition notation and supporting utilities for it.

Another approach is to embed all possible types of messages as dedicated optional fields within the wrapper message. OneOf directive ensures that the wrapper contains only one payload type at once.

```
message MessageWrapper {
    string module = 1;
    string instance = 2;
    int64 ts = 3;
    oneof data {
        BroodNest brood_nest = 4;
        EnvironmentMonitor env_monitor = 5;
        HiveScale hive_scale = 6;
        WeatherStation weather_station = 7;
        DanceDetection dance_detection = 8;
        GNSS gnss = 9;
        PowerSupply power_supply = 10;
        TrafficFlow traffic_flow = 11;
    }
}
```

Such message definition provides strict control over payload structure, however it quickly becomes overcomplicated (the code above shows only part of payloads for encoding sensor readings). Also field naming tends to become confusing, for example the broodnest module has both sensors and actuators which need distinct field names.

A slight benefit of using universal wrapper message is the possibility to process (decode) it at a single component and select appropriate action depending on the payload itself. However taking into account that messages are distributed over hierarchical MQTT topics there is no strict necessity for such a single endpoint feature.

Thus the opposite way of designing messages was selected for implementation: specific messages are designed for all major use-cases, and embedded payload definitions are added as fields in a hierarchical manner. At the current stage dedicated messages are defined for top-level types of messages corresponding to MQTT topic groups, like live stream, alerts, queries, etc. Specific subtypes of messages are implemented as hierarchical fields.

The header field is added to all message types. It contains information about the origin module of the message, its instance identifier (serial number or name) and timestamp. In addition optional tag values can be added for finer message identification (e.g. hardware version or firmware release).

```
message Header {
  string module = 1;
  string instance = 2;
  int64 ts = 3;
  map<string, string> tag = 4;
}
```

An example of a live stream message is shown below. It consists of a header and one of specific payload messages.

```
message LiveStreamItem {
    common.Header header = 1;
    oneof payload {
        sensors.SensorLiveItem sensors = 2;
        actuators.ActuatorLiveItem actuators = 3;
        system.SystemLiveItem system = 4;
        models.ModelLiveItem models = 5;
    }
}
```

The payload messages are further detailed as needed. An example of definitions for sensor messages are as follows.

```
message SensorLiveItem {
    oneof sensors {
        Broodnest broodnest = 1;
        TrafficFlow traffic_flow = 2;
        Scales hive_scales = 3;
        PowerSupply power_supply = 4;
    }
}
```

```
message Broodnest {
    repeated float temp = 1;
    repeated float rh = 2;
    float co2 = 3;
}

message TrafficFlow {
    float bees_in = 1;
    float bees_out = 2;
}

message Scales {
    float weight = 1;
}

message PowerSupply {
    float voltage = 1;
    float current_draw = 2;
    float charge = 3;
}
```

In case if the message hierarchy becomes overly complicated or a message type contains too many subtypes it is possible to move the root level of the messages to the higher more detailed definitions.

Protobuf sniffer utility

Protobuf messages are encoded into binary strings and sent as MQTT payload. However, investigating and debugging such communication is a tedious task. There are tools capable of subscribing for specific MQTT topics and logging transferred messages. Such tools are, however, not able to decode binary payloads.

Thus special sniffer utility is developed which is able to subscribe for specified topics and logs decoded messages in a human readable way. An example of its output is shown below. Also the sniffer utility was used in D3.3 demonstration videos.

```
(ho-interfaces) user@host:~/ho-protocols-sniffer$ python main.py
2023-01-27 11:22:05 INFO    Using provided client credentials
2023-01-27 11:22:05 INFO    Connecting to science.itf.llu.lv:18883
2023-01-27 11:22:05 INFO    Connected to broker as "HO MQTT sniffer sCp6a3dd0d"
2023-01-27 11:22:05 INFO    Subscribed for ho/#
2023-01-27 11:23:25 MESSAGE ho/hiveA/alert/system/testing/abc
header {
  module: "testing"
  instance: "abc"
  ts: 1674811405
}
system {
  power_failure {
  }
}

2023-01-27 11:31:24 MESSAGE ho/hiveA/live/sensors/power_supply/demo1
header {
  module: "power_supply"
  instance: "demo1"
}
sensors {
  power_supply {
    level: High
  }
}
```

Notification service

The main task of this service is to notify users about relevant alerts occurring within the HIVEOPOLIS ecosystem. Depending on the type of alert and specific use-case the target audience can be beekeepers, technical engineers or scientific community. The notification channel highly depends on the audience and its preferences.

For demonstration purposes the notification service was implemented as standalone service which subscribes for all alerts in the global topics:

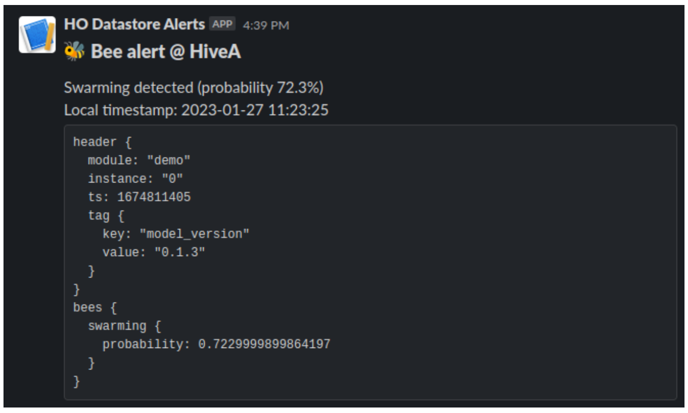
- `ho/+/alert/#`
  "+" sign is a placeholder for client usernames;
  "#" sign allows any suffix on the topics (effectively all types of alerts from any component).

The service has read-only access to specified messages as defined in broker's ACL:

```
user notification_service
topic read ho/+/alert/#
```

The notifications themself are implemented in a form of Slack messages. The service is registered as a Slack application with access to a specific channel. It decodes the incoming MQTT message, converts it into Slack message (applies basic formatting) and posts to the Slack channel via Webhook feature available for registered applications on Slack platform. Similar approach is also available almost for all modern communication platforms.

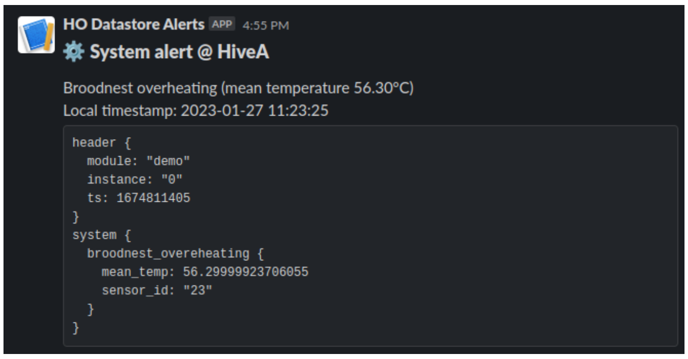
An example of messages as rendered in Slack channel:



HO Datastore Alerts APP 4:39 PM
🐝 **Bee alert @ HiveA**

Swarming detected (probability 72.3%)
Local timestamp: 2023-01-27 11:23:25

```
header {
  module: "demo"
  instance: "0"
  ts: 1674811405
  tag {
    key: "model_version"
    value: "0.1.3"
  }
}
bees {
  swarming {
    probability: 0.7229999899864197
  }
}
```


HO Datastore Alerts APP 4:55 PM

System alert @ HiveA

Broodnest overheating (mean temperature 56.30°C)
Local timestamp: 2023-01-27 11:23:25

```
header {
  module: "demo"
  instance: "0"
  ts: 1674811405
}
system {
  broodnest_overeheating {
    mean_temp: 56.29999923706055
    sensor_id: "23"
  }
}
```

Figure 3: Bee and system alerts rendered as Slack messages

Archive services

There are several applications considered for a live data stream. One is implementing reactive control scenarios when components perform actions immediately upon receiving relevant signals from other components. This type of control is demonstrated in D3.3 with interaction between local components of the Core module. However due to transparent bridging between local and remote systems it is possible to implement similar reactive control involving components hosted in the cloud infrastructure.

Another use of live data stream is maintaining an archive of messages via storing transmitted values into a database. Such a database allows queries for historical values and opens possibilities for more advanced data processing and modelling, such as trend analysis or correlations between different parameters.

The latter scenario was actively investigated during the development of data & programming interfaces. Experiments were performed using available hardware and software solutions. As a source of live data stream beehive monitoring devices were used. The main unit of the monitoring device was a resource constrained microchip (ESP8266), thus emphasising the ability of the developed data & programming interfaces to be used in the embedded environment.

The hardware specifications for the test devices (ESP8266 microchips) are given below:

- Tensilica L106 ultra-low-power 32-bit CPU with clock speed up to 160 MHz;
- available SRAM space < 50 kB;
- external flash size typically 4 MB, but theoretically up to 16 MB;
- integrated Wi-Fi.

The Protobuf schema that was used for testing was described under chapter “Payload formats”. The steps that were required for the device to send data over the developed infrastructure included data encoding according to the defined schema, establishing a connection (using different types of communication technologies – Wi-Fi, 2G cellular network) with the hosted MQTT broker and data publishing.

The monitoring devices on a regular basis (approx. every 30 minutes) reported latest readings from sensors (temperature inside and outside the hive, its weight), as well as internal parameters of the hardware (battery levels of the main and network modules, WiFi connection level). In total 15 hives were monitored during the summer season of 2022. The experiment provided a sufficient live stream of actual values from the devices.

Dedicated live data stream processing services were developed and hosted in the cloud. Experiments with several destination archive databases were conducted. The services connected to the hosted MQTT broker and subscribed to topics dedicated for live stream messages:

- `ho/+/live/#`

The services have read-only access to specified messages as defined in broker's ACL:

```
user archive
topic read ho/+/live/#
```

Upon arriving the messages are decoded and put into a task queue for writing records to a database. Two destination databases were used in parallel during experiments:

- A stand-alone InfluxDB instance hosted in the cloud. Grafana platform is used for data visualisation.
- SAMS data warehouse, which has a dedicated API for data-in requests. The built-in user interface is used for data visualisation.

The experiment with multiple archive databases demonstrates the versatility of the approach. Dedicated services can be fine-tuned for specific types of data or archiving technologies.

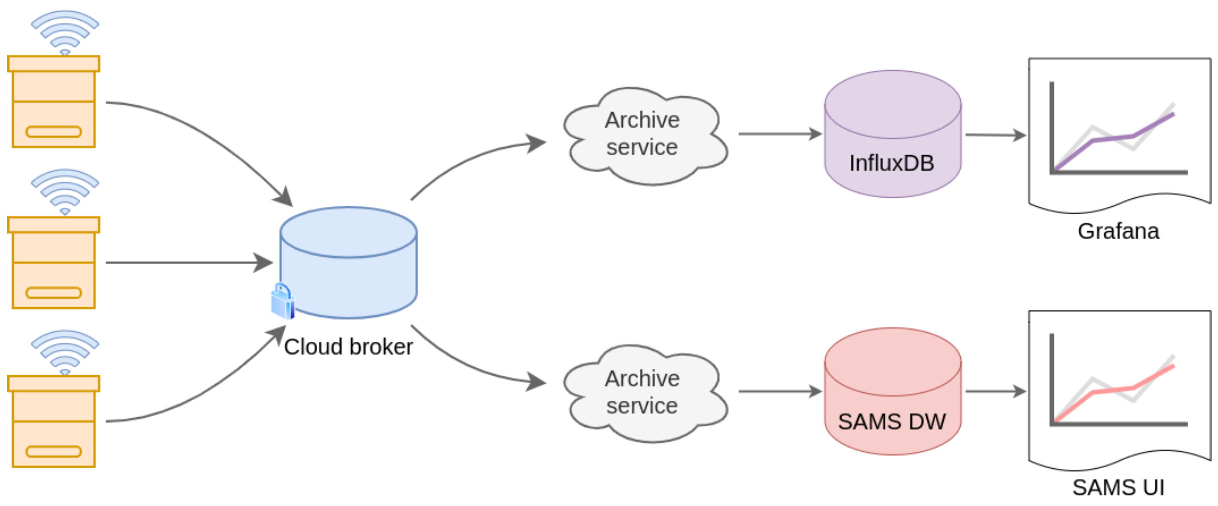


Figure 4: Experimental setup for archive services

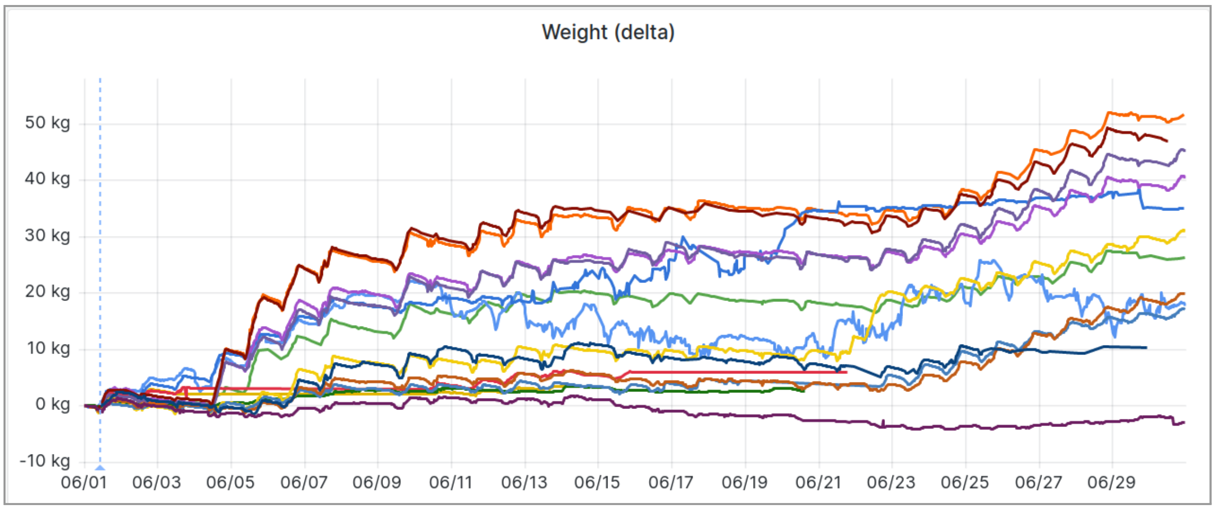


Figure 5: Cumulative weight (kg) of all monitored hives
June 2022, Grafana dashboard (screenshot)

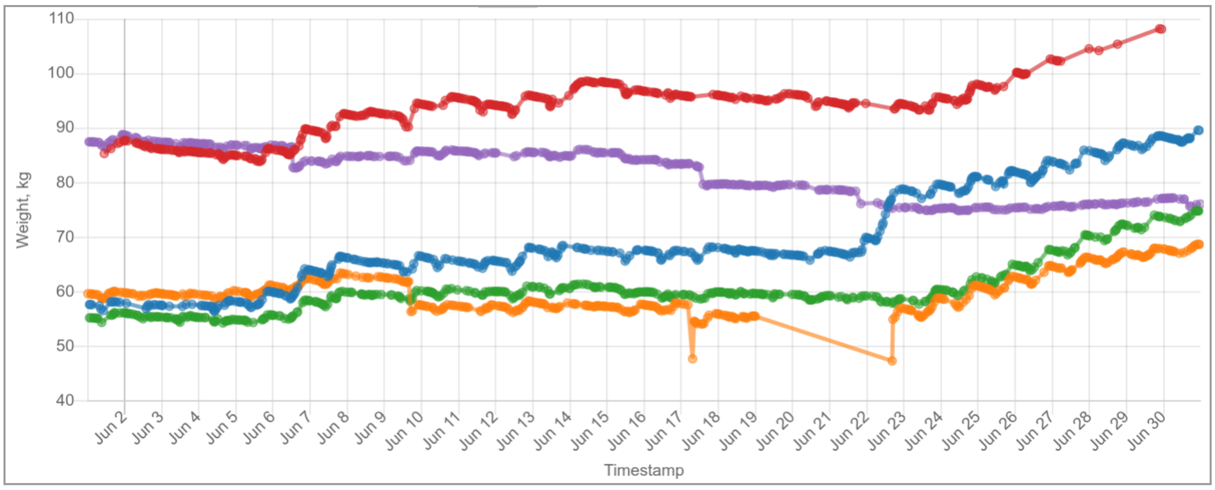


Figure 6: Absolute weight (kg) of the monitored hives in a single apiary
June 2022, SAMS UI dashboard (screenshot)

Overall the experiments demonstrate the benefits of the used distributed architecture: live data stream providers (publishers) are decoupled from data consumers (subscribers). Both sides of such data exchange can be independently modified and extended according to needs. While during experiments the archive services consumed all types of live stream messages, it is also possible to build an archive service for messages from specific modules by subscribing to dedicated MQTT sub-topics.

Query engine service

The purpose of this service is to provide controlled access to the archive data via implementing predefined queries. First of all, the service ensures access authorisation to the archive. For example, the HIVEOPOLIS modules are authorised to request data only about themselves, but dedicated models are permitted to access only limited parameters about all entities. Secondly, the service implements only feasible data queries in a most effective way.

It prevents external components from running unbounded and ineffective queries which might affect the performance of the databases.

As described in the architecture chapter, data queries are implemented using the pair of request-response MQTT topics. The implemented example of the query engine subscribes for all data requests:

- `ho/+/query/+`
  the first "+" sign is a placeholder for username;
  the second "+" sign is a placeholder for request callback ID provided by the sender.

Upon processing the request and results from the database are encoded and sent back to the corresponding topic. An example of request-response topics are as follows:

- `ho/hiveA/query/abc123`
  request from "hiveA" with callback ID "abc123"
- `ho/hiveA/query/abc123/resp`
  response for "hiveA" with callback ID "abc123"

The requesting component is responsible for tracking its requests and corresponding responses. The usual procedure implies the following steps: obtaining request ID (e.g UUID), subscribing for future response, sending the request, upon receiving the response (or by timeout) unsubscribing from the topic.

Query engine is itself is limited only to data query topics as specified in MQTT broker ACL configuration:

```
user query_engine
topic read  ho/+/query/+
topic write ho/+/query/+/resp
```

Depending on the query type the request message contains different parameters. A few examples of data queries are implemented in the service.

Aggregated historical weight

This type of query accepts requests with specified start and end timestamps and in the response returns the aggregated weight for the given period. Corresponding Protobuf definitions are as follows:

```
message HistoricalWeight {
    int64 start_ts = 1;
    int64 end_ts = 2;
}

message HistoricalWeightResult {
    float min_weight = 1;
    float max_weight = 2;
    float mean_weight = 3;
}
```

Actual request-response messages sent via MQTT are as follows (log from the Protobuf sniffer utility):

```
2023-02-01 14:43:35    MESSAGE    ho/hive06/query/c2092076-4ba3-4e12-a58e-3e31da61dff0
header {
  module: "demo"
  instance: "0"
  ts: 1675255415
}
historical_weight {
  start_ts: 1654030800
  end_ts: 1656622800
}

2023-02-01 14:43:35    MESSAGE    ho/hive06/query/c2092076-4ba3-4e12-a58e-3e31da61dff0/resp
historical_weight {
  min_weight: 48.635581970214844
  max_weight: 143.93418884277344
  mean_weight: 123.4678726196289
}
```

Underlying query for Influx DB instance is as follows:

```
data = from(bucket: "eternal")
  |> range(start: {start_ts}, stop: {end_ts})
  |> filter(fn: (r) => r["_measurement"] == "weight" and  r["source"] == "{name}")

data |> min() |> yield(name: "min")
data |> max() |> yield(name: "max")
data |> mean() |> yield(name: "mean")
```

Recent temperature measurements

This type of query provides up to 200 recent temperature data points aggregated depending on the requested resolution. Corresponding Protobuf definitions are as follows:

```
message RecentTemperature {
    enum Resolution {
        Minute = 0;
        Hour = 1;
        Day = 2;
    }
    Resolution resolution = 1;
}

message RecentTemperatureResult {
    message Record {
        int64 ts = 1;
        float min = 2;
        float max = 3;
        float mean = 4;
    }
    int64 start_ts = 1;
    int64 end_ts = 2;
    repeated Record records = 3;
}
```

Request-response MQTT messages are as follows:

```
2023-02-01 16:08:01    MESSAGE    ho/hive06/query/ef58c106-c1e5-47f2-826a-83ed8a252bd9
header {
  module: "demo"
  instance: "0"
  ts: 1675260481
}
recent_temperature {
  resolution: Day
}

2023-02-01 16:08:02    MESSAGE    ho/hive06/query/ef58c106-c1e5-47f2-826a-83ed8a252bd9/resp
recent_temperature {
  start_ts: 1657980481
  end_ts: 1675260481
  records {
    ts: 1658016000
    min: 27.875
    max: 32.875
    mean: 31.646875381469727
  }
  records {
    ts: 1658102400
    min: 23.25
    max: 33.6875
    mean: 30.227394104003906
  }
  < ... more records omitted ... >
}
```

Underlying query for Influx DB instance is as follows:

```
data = from(bucket: "eternal")
    |> range(start: -200{window})
    |> filter(fn: (r) => r["_measurement"] == "temperature" and r["source"] == "{name}")

data_min = data
    |> aggregateWindow(every: 1{window}, fn: min, createEmpty: false)
    |> set(key: "grp", value: "min")
data_max = data
    |> aggregateWindow(every: 1{window}, fn: max, createEmpty: false)
    |> set(key: "grp", value: "max")
data_mean = data
    |> aggregateWindow(every: 1{window}, fn: mean, createEmpty: false)
    |> set(key: "grp", value: "mean")

union(tables: [data_min, data_max, data_mean])
    |> pivot(rowKey: ["_time"], columnKey: ["grp"], valueColumn: "_value")
```

Actuator control

As previously described under “MQTT topic hierarchy”, it is also possible to send specific commands to instruct a certain actuator to perform action. Such a use case was shown in the D3.3 demonstrator. In this particular case an embedded device was subscribed to a command request topic and was listening for incoming commands (either related to the simulation of harvester or evacuation system). Since this time the embedded device acted as subscriber, the incoming messages were decoded and appropriate action was taken (open/close harvester; start/stop evacuation system).

Within the HIVEOPOLIS ecosystem actuators are considered only within hives. Thus there are no cloud services implemented which are listening for commands. However, services for sending the commands are considered. Command protocol is implemented similarly to queries and it relies on a pair of MQTT topics for request-response communication. An example of ACL record is as follows:

```
user harvesting_procedure
```

```
topic write ho/+/cmd/evacuation/demo1/+
topic read  ho/+/cmd/evacuation/demo1/+/resp
topic write ho/+/cmd/harvester/demo1/+
topic read  ho/+/cmd/harvester/demo1/+/resp
```

Corresponding Protobuf messages are implemented as follows:

```
syntax = "proto3";
package ho.cmd;

import "google/protobuf/struct.proto";
import "ho_protocols/common.proto";

message CommandRequest {
    common.Header header = 1;
    oneof command {
        EvacuationStart evac_start = 2;
        EvacuationStop evac_stop = 3;
        HarvesterOpen harvester_open = 4;
        HarvesterClose harvester_close = 5;
    }
}

message CommandResponse {
    common.Header header = 1;
    oneof result {
        ACKResult ack = 2;
        NACKResult nack = 3;
    }

    message ACKResult { }

    message NACKResult {
        enum Reason {
            Unknown = 0;
            RequestError = 1;
            ResponseError = 2;
        }
        Reason reason = 1;
        google.protobuf.Struct details = 2;
    }
}

message EvacuationStart { }
message EvacuationStop { }

message HarvesterOpen { }
message HarvesterClose { }
```

Augmented map service

The augmented map service is implemented as a stand alone component hosted in the cloud which provides data & programming interfaces for spatial information. The service implements data access similar to archive data queries. The pair of MQTT topics is used for request-response communication. The service connects to the hosted MQTT broker and serves requests on following topics:

- `ho/+/map/+/+`
  The “+” signs correspond to placeholders for username, request type and ID.

Appropriate ACL restrictions are configured on the hosted MQTT broker as follows:

```
user map_service
topic read ho/+/map/+/+
topic write ho/+/map/+/+/resp
```

Examples of supported data sources and queries are described in following chapters.

Weather forecast

Augmented map service provides an integration with external weather forecast data sources (as an example OpenWeatherMap is used). It allows weather forecast requests for given GPS coordinates and returns essential results suitable for embedded platforms. In addition the service implements a caching mechanism for subsequent requests.

Protobuf definitions for request-response messages are as follows:

```
message WeatherForecast {
  float lat = 1;
  float lng = 2;
  int32 periods = 3;
}

message WeatherForecastResult {
  message Item {
      int64 ts = 1;
      float temperature = 2;
      float humidity = 3;
      float wind_speed = 4;
      float wind_deg = 5;
  }
  repeated Item item = 1;
}
```

Examples of actual messages as logged by the Protobuf sniffer utility are shown below.

```
2023-02-02 14:38:20 MESSAGE ho/hiveA/map/forecast/a0f160a1-93ed-4c37-aa55-751cb0403541
header {
  module: "demo"
  instance: "0"
  ts: 1675341500
}
weather_forecast {
  lat: 56.650943756103516
  lng: 23.71440315246582
  periods: 2
}

2023-02-02 14:38:20 MESSAGE ho/hiveA/map/forecast/a0f160a1-93ed-4c37-aa55-751cb0403541/resp
weather_forecast {
  item {
    ts: 1675350000
    temperature: 0.6399999856948853
    humidity: 81.0
    wind_speed: 5.380000114440918
    wind_deg: 341.0
  }
  item {
    ts: 1675360800
    temperature: 0.18000000715255737
    humidity: 77.0
    wind_speed: 5.519999980926514
    wind_deg: 343.0
  }
}
```

Harvesting resources model

Augmented map service provides integration with external geographical information systems (GIS). As an example, the Rural Support Service of Latvia (LAD) is used, which provides open access to rural field registry via Web Map Service (WMS) and Web Feature Service (WFS) interfaces. An interactive map of the registry is available at https://karte.lad.gov.lv/.

The registry is used to obtain information about cultivated plants at a requested location, which is further fed into a model for estimating harvesting resources available for bee colonies.

WFS integration

WFS Interface Standard provides an interface allowing requests for geographical features across the web using platform-independent calls (https://www.ogc.org/standard/wfs/).

The LAD registry, among general geometry features, provides information about planted crops and declared area. In order to access available features a special type of request should be composed. WFS interfaces rely on XML data format both for requests and responses.

In general, WFS interfaces should be compatible with each other, however in practice a lot of functionality is vendor specific. The full report about specific WFS interfaces can be obtained via `GetCapabilities` request. The notable parameters are the list of supported endpoints, output formats, implemented operations, filtering capabilities and coordinate reference systems.

```
https://karte.lad.gov.lv/arcgis/services/lauki/MapServer/WFSServer
    ?service=WFS
    &request=GetCapabilities
    &version=2.0.0
```

Geographical features are obtained via `GetFeature` requests. However without additional parameters the WFS interfaces will return the default slice of the database (in case of LAD these are first 2000 records from the north-west corner of map). Thus additional filters should be provided for more useful results. The filters are XML documents specifying how WFS interfaces should filter database records before providing a response, however actual interpretation of the document is WFS vendor specific.

An example of composite XML document for filtering specific types of crops (products) within defined area is as follows:

```
<fes:Filter xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:fes="http://www.opengis.net/fes/2.0">
  <fes:And>
    <fes:Or>
```

```
      <fes:PropertyIsEqualTo>
        <fes:PropertyName>PRODUCT_CODE</fes:PropertyName>
        <fes:Literal>212</fes:Literal>
      </fes:PropertyIsEqualTo>
      <fes:PropertyIsEqualTo>
        <fes:PropertyName>PRODUCT_CODE</fes:PropertyName>
        <fes:Literal>919</fes:Literal>
      </fes:PropertyIsEqualTo>
    </fes:Or>
    <fes:Within>
      <fes:ValueReference>SHAPE</fes:ValueReference>
      <gml:Envelope>
        <gml:lowerCorner>23.65 56.6</gml:lowerCorner>
        <gml:upperCorner>23.85 56.7</gml:upperCorner>
      </gml:Envelope>
    </fes:Within>
  </fes:And>
</fes:Filter>
```

For the harvesting resource model it is needed to filter fields within the reach of a bee colony (circle around the hive with radius of ~3.5 km). Potentially `DWithin` spatial filter can be used for this purpose, however the LAD WFS interface does not support it. Thus similar functionality was implemented via `Intersects` filter, where reference polygon was dynamically generated as a circle. The resulting XML document with filter is as follows:

```
<fes:Filter xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:gml="http://www.opengis.net/gml"
  xmlns:fes="http://www.opengis.net/fes/2.0">
  <fes:Intersects>
    <fes:ValueReference>SHAPE</fes:ValueReference>
    <gml:Polygon>
      <gml:exterior>
        <gml:LinearRing>
          <gml:posList>
            22.88788 56.49753
            <... more points omitted ...>
            22.94787 56.46753
          </gml:posList>
        </gml:LinearRing>
      </gml:exterior>
    </gml:Polygon>
  </fes:Intersects>
</fes:Filter>
```

The response from WFS interfaces is also an XML document with all the details about matched records. A partial example of such a response is shown below. Also the records plotted are shown on figure 7.

```
<?xml version="1.0" encoding="utf-8" ?>
<wfs:FeatureCollection xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:gml="http://www.opengis.net/gml/3.2"
  xmlns:LAD="https://karte.lad.gov.lv/arcgis/services/lauki/MapServer/WFSServer"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  timeStamp="2023-02-02T14:25:13Z" numberMatched="unknown" numberReturned="370">
  <wfs:member>
    <LAD:Lauki gml:id="Lauki.2245203">
      <LAD:OBJECTID>2245203</LAD:OBJECTID>
      <LAD:PERIOD_CODE>2022</LAD:PERIOD_CODE>
      <LAD:PARCEL_ID>16483418</LAD:PARCEL_ID>
      <LAD:PRODUCT_CODE>212</LAD:PRODUCT_CODE>
      <LAD:AID_FORMS>VPM</LAD:AID_FORMS>
      <LAD:AREA_DECLARED>3.20000000</LAD:AREA_DECLARED>
      <LAD:DATA_CHANGED_DATE>2022-05-26T07:47:39</LAD:DATA_CHANGED_DATE>
      <LAD:SHAPE>
        <gml:MultiSurface gml:id="Lauki.2245203.pl" srsName="urn:ogc:def:crs:EPSG::3059">
          < ... geometry definition omitted ... >
        </gml:MultiSurface>
      </LAD:SHAPE>
      <LAD:PRODUCT_DESCRIPTION>Rapsis, ziemas</LAD:PRODUCT_DESCRIPTION>
      <LAD:SHAPE.AREA>31991.51256294</LAD:SHAPE.AREA>
      <LAD:SHAPE.LEN>1103.74115493</LAD:SHAPE.LEN>
    </LAD:Lauki>
  </wfs:member>
  < ... more wfs:member items omitted ... >
</wfs:FeatureCollection>
```

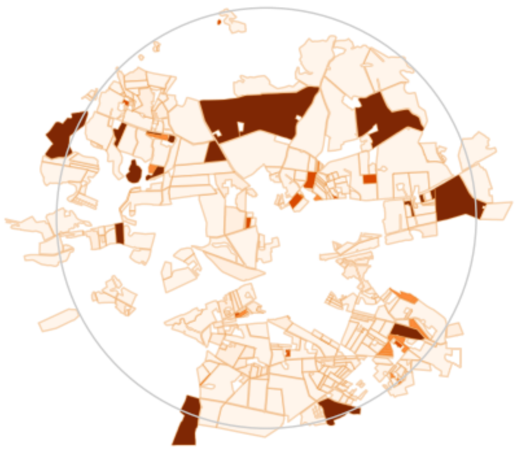


Figure 7: Field polygons around Vecauce loaded from LAD database
(reference circle has a 3.5 km radius).

Additional spatial databases can be integrated into augmented map service if specific information is required.

Model integration

In the next phase the spatial information obtained from the WFS interface should be parsed and used within the harvesting resource estimation model. For demonstration purposes the integrated model uses only information about the plants, however it can be extended by adding more data sources about roads, water bodies, and other spatial features. Models themself are described in detail in D1.1 and are out of scope of the current document.

Protobuf definitions for harvesting request-response messages are as follows:

```
message HarvestingResources {
  float lat = 1;
  float lng = 2;
  int64 ts = 3;
}

message HarvestingResourcesResult {
  message Field {
    string plant_name = 1;
    float amount = 2;
    float dir = 3;
  }
  float overall_index = 1;
  float nectar_index = 2;
  float pollen_index = 3;
  float water_index = 4;
  float road_index = 5;
  repeated Field field = 6;
}
```

Examples of actual messages as logged by the Protobuf sniffer utility are shown below.

```
2023-02-03 11:18:00 MESSAGE ho/hiveA/map/resources/91f113a9-274b-4dc8-8876-8c7120a78ba2
header {
  module: "demo"
  instance: "0"
  ts: 1675415880
}
harvesting_resources {
  lat: 56.467529296875
  lng: 22.887880325317383
  ts: 1675415880
}

2023-02-03 11:18:00 MESSAGE ho/hiveA/map/resources/91f113a9-274b-4dc8-8876-8c7120a78ba2/resp
harvesting_resources {
  overall_index: 338.70550537109375
  field {
    plant_name: "Rapsis, ziemas"
    amount: 110.62000274658203
  }
  field {
    plant_name: "Rapsis, ziemas"
    amount: 51.95000076293945
  }
  < ... more fields omitted ... >
}
```

References

Currier, C. (2022). Protocol buffers. In Mobile Forensics–The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices (pp. 223-260). Cham: Springer International Publishing.

Nastase, L. (2017, May). Security in the internet of things: A survey on application layer protocols. In 2017 21st international conference on control systems and computer science (CSCS) (pp. 659-666). IEEE.

Pradana, M. A., Rakhmatsyah, A., & Wardana, A. A. (2019, September). Flatbuffers implementation on mqtt publish/subscribe communication as data delivery format. In 2019 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI) (pp. 142-146). IEEE.