AIDERS

Deliverable 3.2 Geo-temporal graph database solution

Coordinator Name: **Christos Panayiotou**

Coordinator Email: **christosp@ucy.ac.cy**

Project Name: **Real-time Artificial Intelligence for DEcision support via RPAS data analyticS**

Acronym: **AIDERS**

Grant Agreement: **873240**

Project Website: http://www.kios.ucy.ac.cy/aiders/

Version: **1.0**

Submission Date: **30/06/2021**

Dissemination Level: **Public**

The project has received funding from the European Union Civil Protection Call for proposals UCPM-2019-PP-AG for prevention and preparedness projects in the field of civil protection and marine pollution under grant agreement – 873240– AIDERS.

Funded by European Union Civil Protection

Contents

Executive Summary

Machine Learning (ML) algorithms can provide some advanced solutions to support decision-making by first responders, but the traceability and explainability of decisions of the learned models may still require to keep track of as much information as possible. This challenge therefore calls for the selection of an appropriate data management layer that can expose spatiotemporal data streams to a wide diversity of ML algorithms, while offering a performant and compact storage of acquired information.

This deliverable explains the architecture of the application and the technologies used to manage the data gathered by all the sensors during a rescue mission. It will specifically focus on the two retained databases, Greycat and MinIO, and the internal API we built to interface the databases and the application.

1 Introduction

The choice of Greycat as the core storage technology for the AIDERS project results from the needs of storing the numerous and large data retrieved from the multiple UAVs in operation, for the purpose of traceability and explainability of field decisions. However, standalone Greycat cannot satisfy all of the requirements expected for the project, so that we need to use an additional database for storing multimedia content acquired by UAVs.

In this deliverable, we will therefore present the two database technologies—namely Greycat and MinIO—which have been selected to store the artefacts delivered by UAVs. Then, we will present the API we designed for the AIDERS application to interact easily with this advanced storage backend.

2 Databases

1 Greycat

Overview

Greycat is the main database used as part of the project AIDERS. It aims at storing most of the data collected by the UAVs along the rescue missions and support the execution of the different ML/AI algorithms that will produce insights on the disasters.

Instead of interacting using a query language, like SQL, Greycat provides its own data processing language. We can store and retrieve data that are organized in graphs composed of nodes and edges, but also perform arithmetic operations and eventually implement machine learning algorithms through the tensor objects and their associated operations that are provided. The computations are more effective since the data is made directly by the database engine instead of being fetched and processed by an external tool.

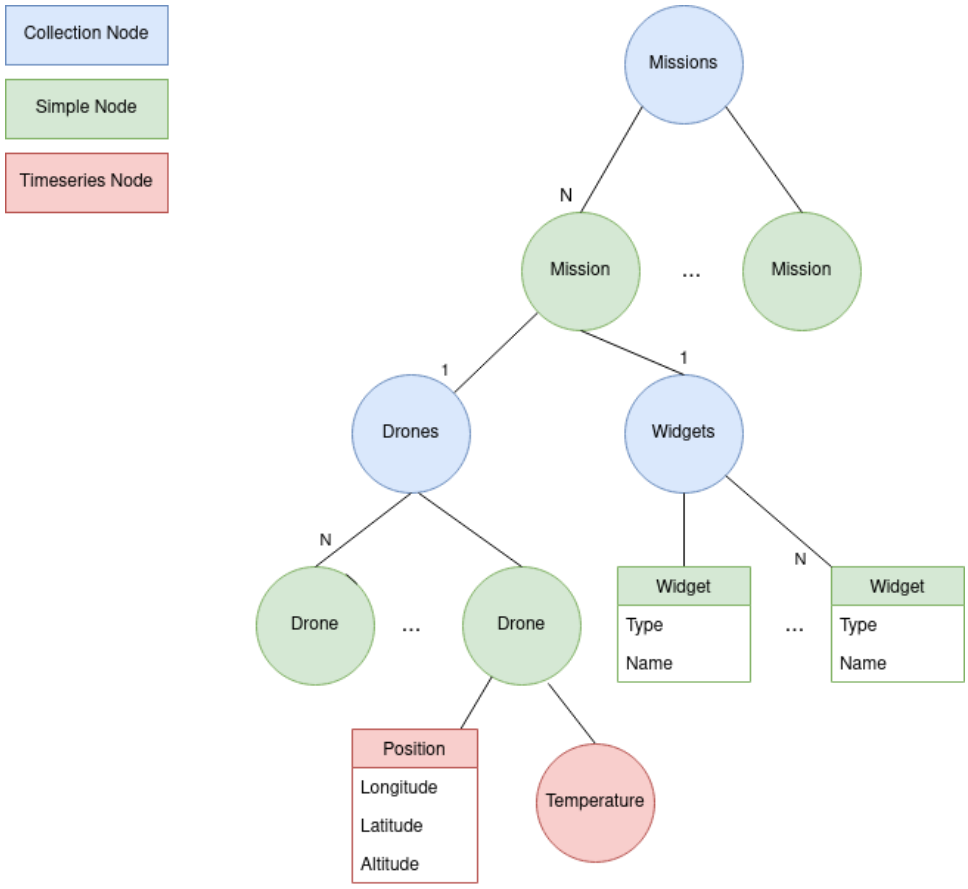
In the following section, we illustrate the application of some of Greycat features for our specific use cases.

Graph structure

With the last versions of Greycat, we cannot use a dedicated base for each mission, so all the missions are gathered in the same database in a root node.

The *missions* node knows about the UAVs that have operated, from where we can get the collected data by each of them. Actually, it contains the geo-localisation data of the drone over time as it is the most basic information to work with, but this scheme is easily expendable to other data, like UAVs orientation and remaining battery.

The *missions* node also contains the widgets, used by the application to render the rescue team insights, and their configuration. This way, end users do not have to recreate all from scratch, if they want to replay or recover from an existing mission.



Timeseries

Greycat supports timeseries natively—*i.e.*, a node can take different values at different points of time. We can create a *nodeTime* object, then we can set its value at a specific timestamp, and we can do this for any number of timestamps we want. Then, when accessing the nodeTime, we can resolve it to a specific value depending on the requested time. There are two ways to accomplish this:

- We can use a time context, so that all resolved nodeTime instances resolve to the value defined at the closest time where a value exists,
- We can use loops that either list all values with their associated times, or sample a value at time intervals.

In our application, we can use this feature to store telemetry data from the UAVs, like geolocation, for example.

Geobox

In a similar manner, it is also possible to define a GeoIndex object. We can then assign values to it at specific coordinates. It can then be resolved to any spatial coordinates, and the value retrieved will be the one defined at the closest point from these coordinates.

Server

Greycat offers the possibility to expose internal functions as a RESTful API through the HTTP protocol. This means that it is possible to deploy a remote instance of Greycat instead of running a dedicated instance to the application. A practical example is the creation of a mission. To do so, we simply use the following code:

```
@expose fn addMission(name: String): i64 {
    missions ?= Map<Mission>::new();
    if (missions.get(name) != null) {
        return 0;
    }
    var missionWidgets: Map<Widget> = Map<Widget>::new();
    var missionDrones: Map<Drone> = Map<Drone>::new();
    missions.set(name,
                 Mission {
                   name: name,
                   widgets: node<Map<Widget>>::new(missionWidgets),
                   drone: node<Map<Drone>>::new(missionDrones),
                   currentDrone: 1
                 });
    return 1;
}
```

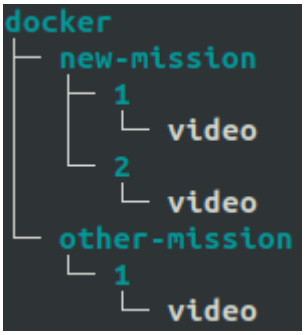
Where, the `@expose` annotation is processed by Greycat to automatically expose the function in the REST API. Then, we run Greycat to serve this function remotely that we can call it through an HTTP request. For example, with `curl`, it would be as easy as executing:

```
curl -d '["New Mission"]' 'http://localhost:8080/addMission'
```

2 MinIO

MinIO is an object storage database that manages data as large objects (also known as BLOB) and abstracts the different filesystems of the underlying storage devices. Using MinIO, we can accommodate the long-term storage of any UAVs artifacts for each mission with a dedicated

bucket for each one, and then store the collected binary data inside the mission's bucket. Even though Greycat is able to store binary data, such as video captured from the UAVs, it is not optimized to manage this kind of data. Therefore, we decided to use MinIO as a secondary database that will take care of these data, but we do not store any other data than Greycat can manage correctly itself.



A potential MinIO database hierarchy: the first level are buckets for each mission, then the different datafiles for each drone that can represent videos, photos etc

MinIO provides several SDK for different languages, like Javascript, which is the one we use in our case, but also C#/.NET, Python, Go, Java, which means this part of the code can be easily reused in case of a rewrite of the application with a different technology. The database was designed to work primarily with AWS S3 storage technology, as it aims to help building hybrid cloud/local systems. Therefore, there is no difference in use between local and cloud storage.

MinIO supports server-side and client-side replication of objects between source and destination buckets. MinIO offers both active-passive (one-way) and active-active (two-way) flavors of the following replication types that can be done automatically through the server configuration or manually using an SDK or the MinIO client. These features allow us to either switch from a local storage to AWS S3 or replicate the database to the Cloud to keep one or several additional saves on a trustworthy Cloud provider.

MinIO can generate URL to query objects through HTTP requests. This is a useful feature for us as we can feed, for instance, the video widget of the application which can then stream a video accessible through this URL, even if the corresponding object has not been completely updated yet, as in the case of a drone broadcasting the video stream of one of its cameras.

MinIO can store multiple versions of a same object and then allow access to these different versions. MinIO is therefore a highly relevant choice for the project, as it allows us to manage large volume of local or streamed binary data.

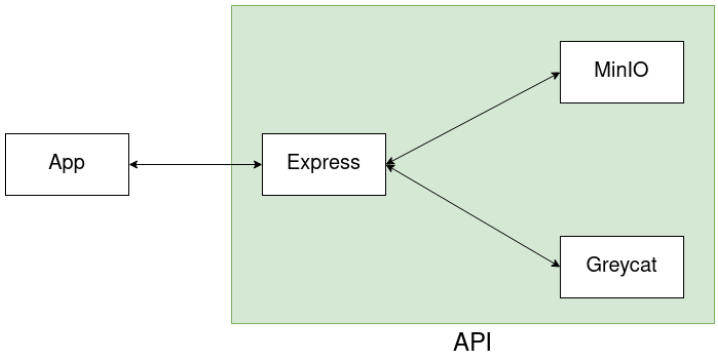
3 RESTful API

We decided to interface the databases with a REST (*REpresentational State Transfer*) API, which is a software architectural style that was created to guide the design and development of the architecture for the Web.

REST abstracts the complexity of interacting with two conceptually different databases, and allows the application to connect through a unique endpoint to non-local instances which are uncoupled from the front application and can run on different environments than the one which the application is running on, which can be incompatible with the different components the API is made of—*i.e.*, Greycat does not run on Windows at the time of publishing this deliverable.

The API is composed of 3 parts: Express, which is a lightweight flexible Node.js web application framework, and the 2 databases, Greycat and MinIO.

The Express server waits for requests coming from the application, then translates these requests and forwards them to Greycat and/or MinIO depending on the necessary actions to execute. The databases reply back to Express with the result of the requests and send them back to the application.



Architecture overview of the AIDERS database solution.

We fetch the API through a set of URIs representing the resources to query, with different HTTP methods depending on which action on these resources is wanted. For example:
- A GET request on `/missions` returns the list of missions in the Greycat database;
- A POST request on `/missions` adds a mission in the database;
- A PUT request on `/missions/:id` modifies the mission identified by the id;
- A DELETE request on `/missions/:id` removes the qualified mission from the databases

The same scheme applies to the widgets of the missions. This way we can create, get, modify, and delete all kinds of data available for both Greycat and MinIO. Actually, the API also

provides access to one recorded video for each UAV, for testing purposes as the format of the data can vary, as well as access and storage of their geo-localisation data.

This interface also simplifies our code from the front side. For instance, to create a mission, the application sends a unique request to Express, then the server deals with the responsibility to request Greycat to create an entry for the new mission and also to request MinIO to create a new bucket for this mission as well.

Geo-localisation data from CSV files are loaded using the application. Since UAVs can transmit and store different types of video streams, hence video files will be received by the MinIO database according to the file type stored.

With this architecture, we can store larger amounts of raw data like video or image data in the MinIO object storage, and store metadata that describes those in the greycat database, that can be used by machine learning algorithms.

Endpoint example

POST `/api/missions/{mission}/geoDrone`
Returns a list of positions taken by a drone with their associated timestamps, for every drone in a mission. Parameter: *mission* in the path is the ID of the mission we want to check
Reponse example:


```
[{
  "_type":"project.DroneRes",
  "droneID":2,
  "geopos":[{
      "_type":"core.Tuple",
      "x":{
        "_type":"core.time",
        "epoch":1592229964,
        "us":0
      },
      "y":{
        "_type":"geo.geo",
        "lat":35.145460183,
        "lng":33.415875169
      }
    }
  ]
}]
```

Conclusion

In this deliverable, we presented Greycat, the data organization and the useful features of the database, MinIO and its features that made it an important supplementary database to Greycat, that make the proposed API useful for the holistic collection, processing and retrieval of data for AI algorithms in the needs of disaster management.