vantage6

A. van Gestel, B. van Beusekom, D. Smits, F. Martin, J. van Soest,

Mar 22, 2023

CONTENTS

CHAPTER

ONE

WHAT IS VANTAGE6?

Vantage6 stands for pri**va**cy preservi**n**g federa**t**ed le**a**rnin**g** infrastructur**e** for **s**ecure **i**nsight e**x**change.

The project is inspired by the Personal Health Train (PHT) concept. In this analogy vantage6 is the *tracks* and *stations*. Compatible algorithms are the *trains*, and computation tasks are the *journey*. Vantage6 is completely open source under the Apache License.

What vantage6 does:

- delivering algorithms to data stations and collecting their results
- managing users, organizations, collaborations, computation tasks and their results
- providing control (security) at the data-stations to their owners

What vantage6 does *not* (yet) do:

- formatting the data at the data station
- aligning data across the data stations (for the vertical partitioned use case)

The vantage6 infrastructure is designed with three fundamental functional aspects of federated learning.

1. **Autonomy**. All involved parties should remain independent and autonomous.
2. **Heterogeneity**. Parties should be allowed to have differences in hardware and operating systems.
3. **Flexibility**. Related to the latter, a federated learning infrastructure should not limit the use of relevant data.

CHAPTER

TWO

OVERVIEW OF THIS DOCUMENTATION

This documentation space consists of the following main sections:

- **Introduction** → *You are here now*
- *User guide* → *How to use vantage6 as a researcher*
- *Node admin guide* → *How to install and configure vantage6 nodes*
- *Server admin guide* → *How to configure and deploy vantage6 servers*
- *Technical Docs* (Under construction) → *Implementation details of the vantage6 platform*
- *Developer community* → *How to collaborate on the development of the vantage6 infrastructure*
- *Algorithm Development* → *Develop algorithms that are compatible with vantage6*
- *Glossary* → *A dictionary of common terms used in these docs*
- *Release notes* → *Log of what has been released and when*

CHAPTER

THREE

VANTAGE6 RESOURCES

This is a - non-exhaustive - list of vantage6 resources.

Documentation

- docs.vantage6.ai → *This documentation.*
- vantage6.ai → *vantage6 project website*
- Academic papers → *Technical insights into vantage6*

Source code

- vantage6 → *Contains all components (and the python-client).*
- Planning → *Contains all features, bugfixes and feature requests we are working on. To submit one yourself, you can create a* new issue.

Community

- Discord → *Chat with the vantage6 community*
- *Community meetings* → *Bi-monthly developer community meeting*

CHAPTER

FOUR

INDEX

4.1 Concepts

4.1.1 Architecture

In vantage6, a **client** can pose a question to the **server**, which is then delivered as an **algorithm** to the **node** (Fig. 4.1). When the algorithm completes, the node sends the results back to the client via the server. An algorithm may be enabled to communicate directly with twin algorithms running on other nodes.

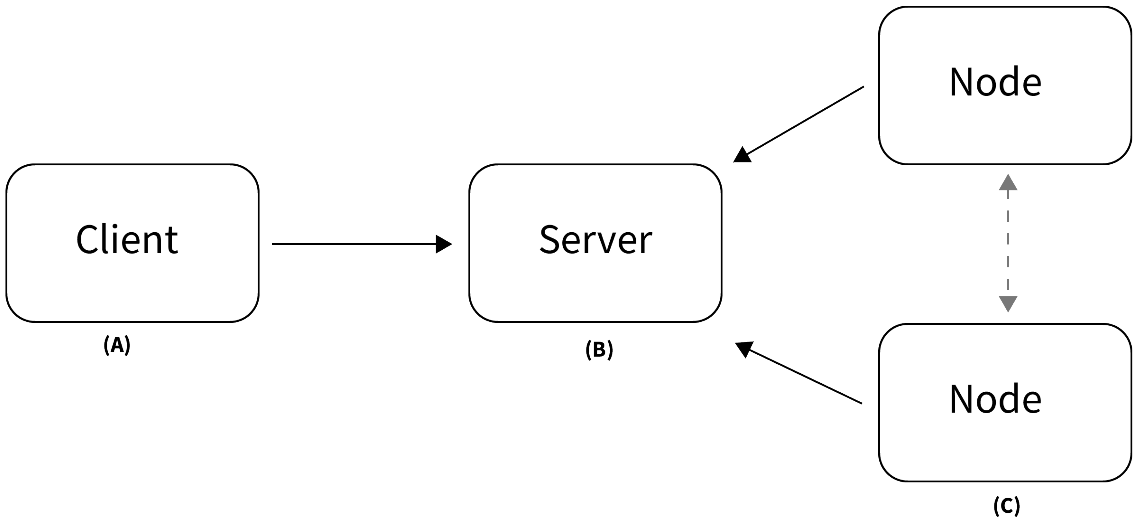


Fig. 4.1: Vantage6 has a client-server architecture. (A) The client is used by the researcher to create computation requests. It is also used to manage users, organizations and collaborations. (B) The server contains users, organizations, collaborations, tasks and their results. (C) The nodes have access to data and handle computation requests from the server.

Conceptually, vantage6 consists of the following parts:

- A (central) **server** that coordinates communication with clients and nodes. The server is in charge of processing tasks as well as handling administrative functions such as authentication and authorization.
- One or more **node(s)** that have access to data and execute algorithms
- **Users** (i.e. researchers or other applications) that request computations from the nodes via the client

- **Organizations** that are interested in collaborating. Each user belongs to one of these organizations.
- A **Docker registry** that functions as database of algorithms

On a technical level, vantage6 may be seen as a container orchestration tool for privacy preserving analyses. It deploys a network of containerized applications that together ensure insights can be exchanged without sharing record-level data.

4.1.2 Entities

There are several entities in vantage6, such as users, organizations, tasks, etc. The following statements should help you understand their relationships.

- A **collaboration** is a collection of one or more **organizations**.
- For each collaboration, each participating organization needs a **node** to compute tasks.
- Each organization can have **users** who can perform certain actions.
- The permissions of the user are defined by the assigned **rules**.
- It is possible to collect multiple rules into a **role**, which can also be assigned to a user.
- Users can create **tasks** for one or more organizations within a collaboration.
- A task should produce a **result** for each organization involved in the task.

The following schema is a *simplified* version of the database:

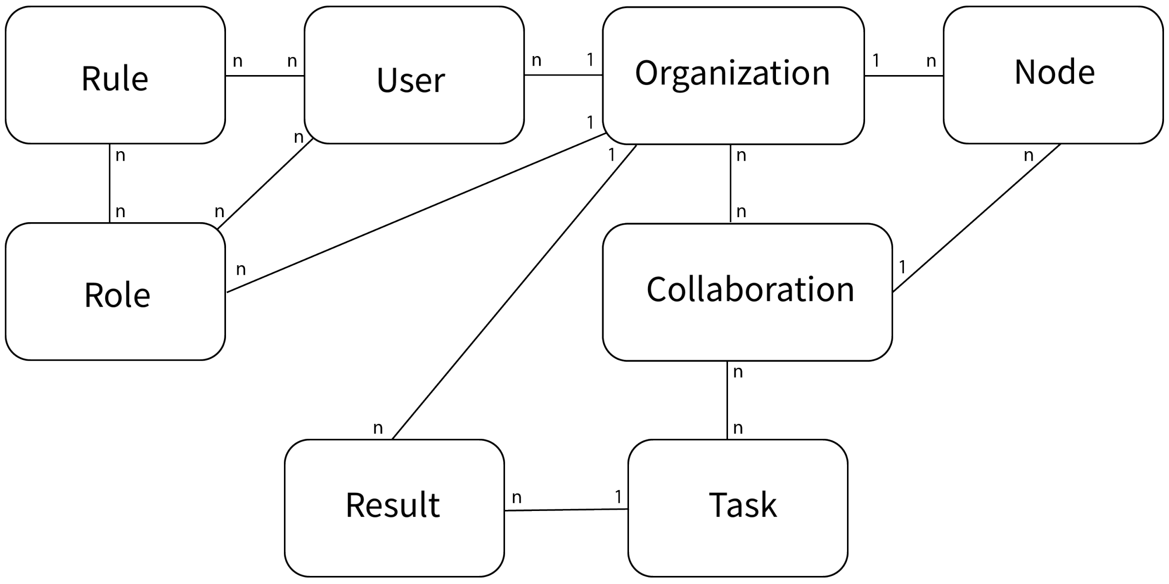


Fig. 4.2: Simplified database model

4.1.3 Network Actors

As we saw in Figure Fig. 4.1, vantage6 consists of a central server, a number of nodes and a client. This section explains in some more detail what these network actors are responsible for.

4.1.3.1 Server

Note: When we refer to the server, this is not just the *vantage6-server*, but also other infrastructure components that the vantage6 server relies on.

The server is responsible for coordinating all communication in the vantage6 network. It consists of several components:

vantage6 server
: Contains the users, organizations, collaborations, tasks and their results. It handles authentication and authorization to the system and is the central point of contact for clients and nodes. .. todo For more details see **`vantage6-server`_**.

Docker registry
: Contains algorithms stored in Images which can be used by clients to request a computation. The node will retrieve the algorithm from this registry and execute it.

VPN server (optionally)
: If algorithms need to be able to engage in peer-to-peer communication, a VPN server can be set up to help them do so. This is usually the case when working with MPC, and is also often required for machine learning applications.

RabbitMQ message queue (optionally)
: The *vantage6 server* uses the socketIO protocol to communicate between server, nodes and clients. If there are multiple instances of the vantage6 server, it is important that the messages are communicated to all relevant actors, not just the ones that a certain server instance is connected to. RabbitMQ is therefore used to synchronize the messages between multiple *vantage6 server* instances.

4.1.3.2 Data Station

vantage6 node
: The node is responsible for executing the algorithms on the **local data**. It protects the data by allowing only specified algorithms to be executed after verifying their origin. The **node** is responsible for picking up the task, executing the algorithm and sending the results back to the server. The node needs access to local data. For more details see the *technical documentation of the node*.

database
: The database may be in any format that the algorithms relevant to your use case support. There is tooling available for CSV, Parquet and SPARQL. There are other data-adapters (e.g. OMOP and FHIR) in development.

4.1.3.3 User or Application

A user or application interacts with the *vantage6-server*. They can create tasks and retrieve their results, or manage entities at the server (i.e. creating or editing users, organizations and collaborations). This can be done using clients or via the user interface.

4.1.4 End to end encryption

Encryption in vantage6 is handled at organization level. Whether encryption is used or not, is set at collaboration level. All the nodes in the collaboration need to agree on this setting. You can enable or disable encryption in the node configuration file, see the example in *All configuration options*.

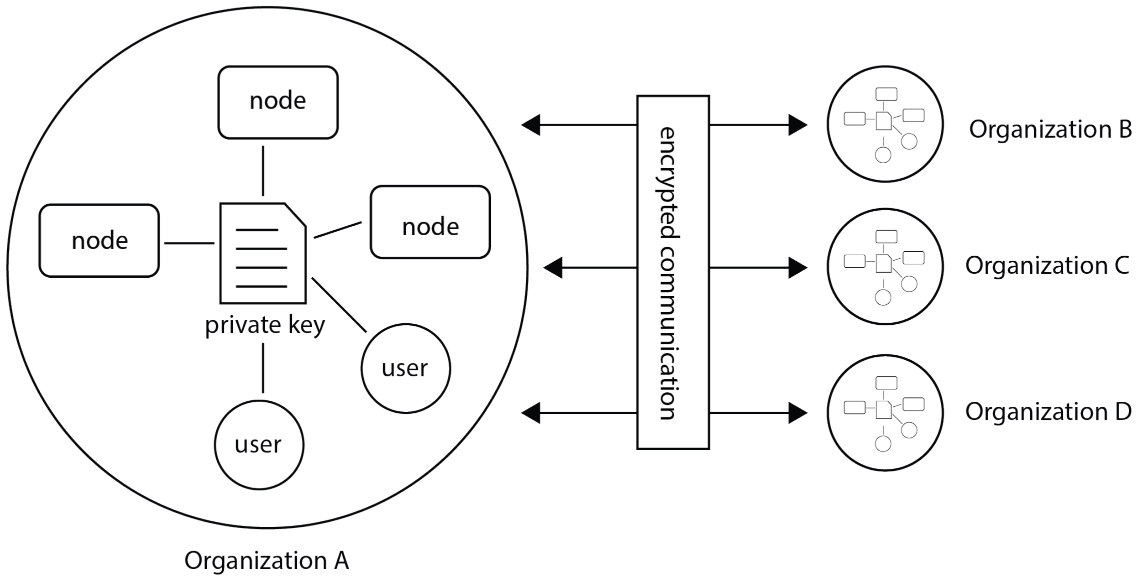


Fig. 4.3: Encryption takes place between organizations therefore all nodes and users from the a single organization should use the same private key.

The encryption module encrypts data so that the server is unable to read communication between users and nodes. The only messages that go from one organization to another through the server are computation requests and their results. Only the algorithm input and output are encrypted. Other metadata (e.g. time started, finished, etc), can be read by the server.

The encryption module uses RSA keys. The public key is uploaded to the vantage6-server. Tasks and other users can use this public key (this is automatically handled by the python-client and R-client) to send messages to the other parties.

Note: The RSA key is used to create a shared secret which is used for encryption and decryption of the payload.

When the node starts, it checks that the public key stored at the server is derived from the local private key. If this is not the case, the node will replace the public key at the server.

Warning: If an organization has multiple nodes and/or users, they must use the same private key.

In case you want to generate a new private key, you can use the command `vnode create-private-key`. If a key already exists at the local system, the existing key is reused (unless you use the `--force` flag). This way, it is easy to configure multiple nodes to use the same key.

It is also possible to generate the key yourself and upload it by using the endpoint `https://SERVER[/api_path]/organization/<ID>`.

4.2 User guide

In this section of the documentation, we explain how you can interact with vantage6 servers and nodes as a user.

There are four ways in which you can interact with the central server: the *User interface* (UI), the *Python client*, the *R client*, and the *API*. In the sections below, we describe how to use each of these methods, and what you need to install (if anything).

For most use cases, we recommend to use the *User interface*, as this requires the least amount of effort. If you want to automate your workflow, we recommend using the *Python client*.

Warning: Note that for some algorithms, tasks cannot yet be created using the UI, or the results cannot be retrieved. This is because these algorithms have Python-specific datatypes that cannot be decoded in the UI. In this case, you will need to use the Python client to create the task and read the results.

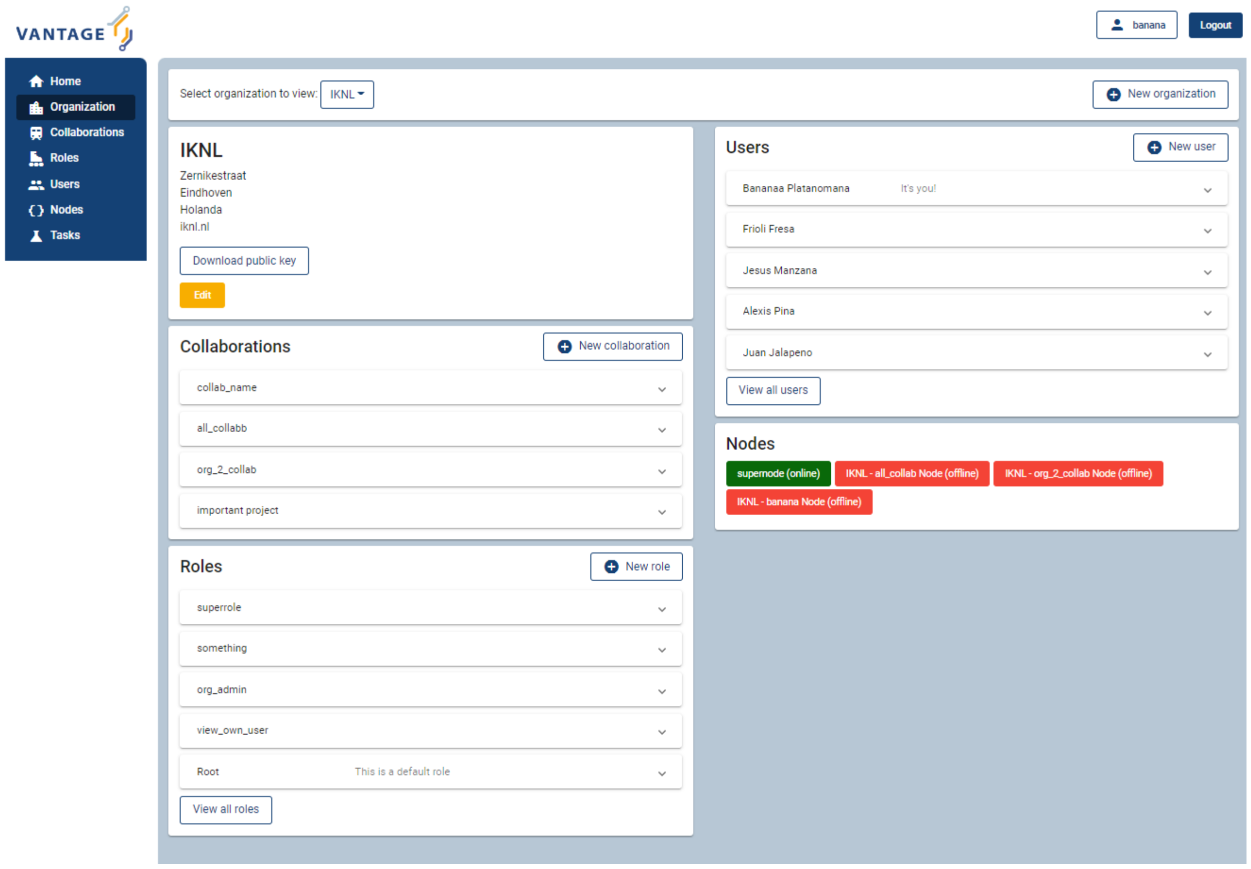
Warning: Depending on your algorithm it *may* be required to use a specific language to post a task and retrieve the results. This could happen when the output of an algorithm contains a language specific datatype and or serialization.

4.2.1 User interface

The User Interface (UI) is a website where you can login with your vantage6 user account. Which website this is, depends on the vantage6 server you are using. If you are using the Petronas server, go to https://portal.petronas.vantage6.ai and login with your user account.

Using the UI should be relatively straightforward. There are buttons that should help you e.g. create a task or change your password. If anything is unclear, please contact us via Discord.

Note: If you are a server administrator and want to set up a user interface, see *this section* on deploying a UI.



Select organization to view: IKNL

IKNL
Zernikestraat
Eindhoven
Holanda
iknl.nl

Collaborations
collab_name
all_collabb
org_2_collab
important project

Roles
superrole
something
org_admin
view_own_user
Root This is a default role

Users
Bananaa Platanomana It's you!
Frioli Fresa
Jesus Manzana
Alexis Pina
Juan Jalapeno

Nodes
supernode (online)
IKNL - all_collab Node (offline)
IKNL - org_2_collab Node (offline)
IKNL - banana Node (offline)

Fig. 4.4: Screenshot of the vantage6 UI

4.2.2 Python client

The Python client is the recommended way to interact with the vantage6 server for tasks that you want to automate. It is a Python library that facilitates communication with the vantage6 server, e.g. by encrypting and decrypting data for tasks for you.

The Python client aims to completely cover the vantage6 server communication. It can create computation tasks and collect their results, manage organizations, collaborations, users, etc. Under the hood, the Python client talks to the server API to achieve this.

4.2.2.1 Requirements

You need Python to use the Python client. We recommend using Python 3.10, as the client has been tested with this version. For higher versions, it may be difficult to install the dependencies.

Warning: If you use a vantage6 version older than 3.8.0, you should use Python 3.7 instead of Python 3.10.

4.2.2.2 Install

It is important to install the Python client with the same version as the vantage6 server you are talking to. Check your server version by going to `https://<server_url>/version` (e.g. *https://petronas.vantage6.ai/version* or *http://localhost:5000/api/version*) to find its version.

Then you can install the `vantage6-client` with:

```
pip install vantage6==<version>
```

where you add the version you want to install. You may also leave out the version to install the most recent version.

4.2.2.3 Use

First, we give an overview of the client. From the section *Authentication* onwards, there is example code of how to login with the client, and then create organizations, tasks etc.

Overview

The Python client contains groups of commands per resource type. For example, the group `client.user` has the following commands:

- `client.user.list()`: list all users
- `client.user.create(username, password, ...)`: create a new user
- `client.user.delete(<id>)`: delete a user
- `client.user.get(<id>)`: get a user

You can see how to use these methods by using `help(...)` , e.g. `help(client.task.create)` will show you the parameters needed to create a new user:

```
help(client.task.create)
#Create a new task
#
#    Parameters
#    ----------
#    collaboration : int
#        Id of the collaboration to which this task belongs
#    organizations : list
#        Organization ids (within the collaboration) which need
#        to execute this task
#    name : str
#        Human readable name
#    image : str
#        Docker image name which contains the algorithm
#    description : str
#        Human readable description
#    input : dict
#        Algorithm input
#    data_format : str, optional
#        IO data format used, by default LEGACY
#    database: str, optional
#        Name of the database to use. This should match the key
#        in the node configuration files. If not specified the
#        default database will be tried.
#
#    Returns
#    -------
#    dict
#        Containing the task information
```

The following groups (related to the *Components*) of methods are available. They usually have `list()`, `create()`, `delete()` and `get()` methods attached - except where they are not relevant (for example, a rule that gives a certain permission cannot be deleted).

- `client.user`
- `client.organization`
- `client.rule`
- `client.role`
- `client.collaboration`
- `client.task`
- `client.result`
- `client.node`

Finally, the class `client.util` contains some utility functions, for example to check if the server is up and running or to change your own password.

Authentication

This section and the following sections introduce some minimal examples for administrative tasks that you can perform with our *Python client*. We start by authenticating.

To authenticate, we create a config file to store our login information. We do this so we do not have to define the `server_url`, `server_port` and so on every time we want to use the client. Moreover, it enables us to separate the sensitive information (login details, organization key) that you do not want to make publicly available, from other parts of the code you might write later (e.g. on submitting particular tasks) that you might want to share publicly.

```
# config.py

server_url = "https://MY VANTAGE6 SERVER" # e.g. https://petronas.vantage6.ai or
                                          # http://localhost for a local dev server
server_port = 443 # This is specified when you first created the server
server_api = "" # This is specified when you first created the server

username = "MY USERNAME"
password = "MY PASSWORD"

organization_key = "FILEPATH TO MY PRIVATE KEY" # This can be empty if you do not want␣
↪to set up encryption
```

Note that the `organization_key` should be a filepath that points to the private key that was generated when the organization to which your login belongs was first created (see *Creating an organization*).

Then, we connect to the vantage 6 server by initializing a Client object, and authenticating

```
from vantage6.client import Client

# Note: we assume here the config.py you just created is in the current directory.
# If it is not, then you need to make sure it can be found on your PYTHONPATH
import config

# Initialize the client object, and run the authentication
client = Client(config.server_url, config.server_port, config.server_api,
                verbose=True)
client.authenticate(config.username, config.password)

# Optional: setup the encryption, if you have an organization_key
client.setup_encryption(config.organization_key)
```

Note: Above, we have added `verbose=True` as additional argument when creating the Client(…) object. This will print much more information that can be used to debug the issue.

Creating an organization

After you have authenticated, you can start generating resources. The following also assumes that you have a login on the Vantage6 server that has the permissions to create a new organization. Regular end-users typically do not have these permissions (typically only administrators do); they may skip this part.

The first (optional, but recommended) step is to create an RSA keypair. A keypair, consisting of a private and a public key, can be used to encrypt data transfers. Users from the organization you are about to create will only be able to use encryption if such a keypair has been set up and if they have access to the private key.

```
from vantage6.common import warning, error, info, debug, bytes_to_base64s
from vantage6.client.encryption import RSACryptor
from pathlib import Path

# Generated a new private key
# Note that the file below doesn't exist yet: you will create it
private_key_filepath = r'/path/to/private/key'
private_key = RSACryptor.create_new_rsa_key(Path(private_key_filepath))

# Generate the public key based on the private one
public_key_bytes = RSACryptor.create_public_key_bytes(private_key)
public_key = bytes_to_base64s(public_key_bytes)
```

Now, we can create an organization

```
client.organization.create(
    name = 'The_Shire',
    address1 = '501 Buckland Road',
    address2 = 'Matamata',
    zipcode = '3472',
    country = 'New Zealand',
    domain = 'the_shire.org',
    public_key = public_key   # use None if you haven't set up encryption
)
```

Users can now be created for this organization. Any users that are created and who have access to the private key we generated above can now use encryption by running

```
client.setup_encryption('/path/to/private/key')
# or, if you don't use encryption
client.setup_encryption(None)
```

after they authenticate.

Creating a collaboration

Here, we assume that you have a Python session with an authenticated Client object, as created in *Authentication*. We also assume that you have a login on the Vantage6 server that has the permissions to create a new collaboration (regular end-users typically do not have these permissions, this is typically only for administrators).

A collaboration is an association of multiple organizations that want to run analyses together. First, you will need to find the organization id's of the organizations you want to be part of the collaboration.

```
client.organization.list(fields=['id', 'name'])
```

Once you know the id's of the organizations you want in the collaboration (e.g. 1 and 2), you can create the collaboration:

```
collaboration_name = "fictional_collab"
organization_ids = [1,2] # the id's of the respective organizations
client.collaboration.create(name = collaboration_name,
                            organizations = organization_ids,
                            encrypted = True)
```

Note that a collaboration can require participating organizations to use encryption, by passing the `encrypted = True` argument (as we did above) when creating the collaboration. It is recommended to do so, but requires that a keypair was created when *Creating an organization* and that each user of that organization has access to the private key so that they can run the `client.setup_encryption(...)` command after *Authentication*.

Registering a node

Here, we again assume that you have a Python session with an authenticated Client object, as created in *Authentication*, and that you have a login that has the permissions to create a new node (regular end-users typically do not have these permissions, this is typically only for administrators).

A node is associated with both a collaboration and an organization (see *Components*). You will need to find the collaboration and organization id's for the node you want to register:

```
client.organization.list(fields=['id', 'name'])
client.collaboration.list(fields=['id', 'name'])
```

Then, we register a node with the desired organization and collaboration. In this example, we create a node for the organization with id 1 and collaboration with id 1.

```
# A node is associated with both a collaboration and an organization
organization_id = 1
collaboration_id = 1
api_key = client.node.create(collaboration = collaboration_id, organization =␣
↪organization_id)
print(f"Registered a node for collaboration with id {collaboration_id}, organization␣
↪with id {organization_id}. The API key that was generated for this node was {api_key}")
```

Remember to save the `api_key` that is returned here, since you will need it when you *Configure* the node.

Creating a task

Preliminaries

Here we assume that

- you have a Python session with an authenticated Client object, as created in *Authentication*.
- you already have the algorithm you want to run available as a container in a docker registry (see here for more details on developing your own algorithm)
- the nodes are configured to look at the right database

In this manual, we'll use the averaging algorithm from harbor2.vantage6.ai/demo/average, so the second requirement is met. This container assumes a comma-separated (*.csv) file as input, and will compute the average over one of the named columns. We'll assume the nodes in your collaboration have been configured to look at a comma-separated database, i.e. their config contains something like

```
databases:
    default: /path/to/my/example.csv
    my_other_database: /path/to/my/example2.csv
```

so that the third requirement is also met. As an end-user running the algorithm, you'll need to align with the node owner about which database name is used for the database you are interested in. For more details, see how to *Configure* your node.

Determining which collaboration / organizations to create a task for

First, you'll want to determine which collaboration to submit this task to, and which organizations from this collaboration you want to be involved in the analysis

```
>>> client.collaboration.list(fields=['id', 'name', 'organizations'])
[
 {'id': 1, 'name': 'example_collab1',
 'organizations': [
     {'id': 2, 'link': '/api/organization/2', 'methods': ['GET', 'PATCH']},
     {'id': 3, 'link': '/api/organization/3', 'methods': ['GET', 'PATCH']},
     {'id': 4, 'link': '/api/organization/4', 'methods': ['GET', 'PATCH']}
 ]}
]
```

In this example, we see that the collaboration called example_collab1 has three organizations associated with it, of which the organization id's are 2, 3 and 4. To figure out the names of these organizations, we run:

```
>>> client.organization.list(fields=['id', 'name'])
[{'id': 1, 'name': 'root'}, {'id': 2, 'name': 'example_org1'},
 {'id': 3, 'name': 'example_org2'}, {'id': 4, 'name': 'example_org3'}]
```

i.e. this collaboration consists of the organizations example_org1 (with id 2), example_org2 (with id 3) and example_org3 (with id 4).

Creating a task that runs the master algorithm

Now, we have two options: create a task that will run the master algorithm (which runs on one node and may spawns subtasks on other nodes), or create a task that will (only) run the RPC methods (which are run on each node). Typically, the RPC methods only run the node local analysis (e.g. compute the averages per node), whereas the master algorithms performs aggregation of those results as well (e.g. starts the node local analyses and then also computes the overall average). First, let us create a task that runs the master algorithm of the harbor2.vantage6.ai/demo/average container

```
input_ = {'method': 'master',
          'kwargs': {'column_name': 'age'},
          'master': True}

average_task = client.task.create(collaboration=1,
                                  organizations=[2,3],
                                  name="an-awesome-task",
                                  image="harbor2.vantage6.ai/demo/average",
                                  description='',
```

(continues on next page)

(continued from previous page)

```
                                  input=input_,
                                  data_format='json')
```

Note that the `kwargs` we specified in the `input_` are specific to this algorithm: this algorithm expects an argument `column_name` to be defined, and will compute the average over the column with that name. Furthermore, note that here we created a task for collaboration with id `1` (i.e. our `example_collab1`) and the organizations with id `2` and `3` (i.e. `example_org1` and `example_org2`). I.e. the algorithm need not necessarily be run on *all* the organizations involved in the collaboration. Finally, note that `client.task.create()` has an optional argument called `database`. Suppose that we would have wanted to run this analysis on the database called `my_other_database` instead of the `default` database, we could have specified an additional `database = 'my_other_database'` argument. Check `help(client.task.create)` for more information.

Creating a task that runs the RPC algorithm

You might be interested to know output of the RPC algorithm (in this example: the averages for the 'age' column for each node). In that case, you can run only the RPC algorithm, omitting the aggregation that the master algorithm will normally do:

```
input_ = {'method': 'average_partial',
          'kwargs': {'column_name': 'age'},
          'master': False}

average_task = client.task.create(collaboration=1,
                                  organizations=[2,3],
                                  name="an-awesome-task",
                                  image="harbor2.vantage6.ai/demo/average",
                                  description='',
                                  input=input_,
                                  data_format='json')
```

Inspecting the results

Of course, it will take a little while to run your algorithm. You can use the following code snippet to run a loop that checks the server every 3 seconds to see if the task has been completed:

```
print("Waiting for results")
task_id = average_task['id']
task_info = client.task.get(task_id)
while not task_info.get("complete"):
    task_info = client.task.get(task_id, include_results=True)
    print("Waiting for results")
    time.sleep(3)

print("Results are ready!")
```

When the results are in, you can get the result_id from the task object:

```
result_id = task_info['id']
```

and then retrieve the results

```
result_info = client.result.list(task=result_id)
```

The number of results may be different depending on what you run, but for the master algorithm in this example, we can retrieve it using:

```
>>> result_info['data'][0]['result']
{'average': 53.25}
```

while for the RPC algorithm, dispatched to two nodes, we can retrieve it using

```
>>> result_info['data'][0]['result']
{'sum': 253, 'count': 4}
>>> result_info['data'][1]['result']
{'sum': 173, 'count': 4}
```

4.2.3 R client

Warning: We discourage the use of the R client. It is not actively maintained and is not fully implemented. It can not (yet) be used to manage resources, such as creating and deleting users and organizations.

4.2.3.1 Install

You can install the R client by running:

```
devtools::install_github('IKNL/vtg', subdir='src')
```

4.2.3.2 Use

The R client can only create tasks and retrieve their results.

Initialization of the R client can be done by:

```
setup.client <- function() {
  # Username/password should be provided by the administrator of
  # the server.
  username <- "username@example.com"
  password <- "password"

  host <- 'https://petronas.vantage6.ai:443'
  api_path <- ''

  # Create the client & authenticate
  client <- vtg::Client$new(host, api_path=api_path)
  client$authenticate(username, password)

  return(client)
}

# Create a client
client <- setup.client()
```

Then, this client can be used for the different algorithms. Refer to the README in the repository on how to call the algorithm. Usually this includes installing some additional client-side packages for the specific algorithm you are using.

Example

This example shows how to run the vantage6 implementation of a federated Cox Proportional Hazard regression model. First you need to install the client side of the algorithm by:

```
devtools::install_github('iknl/vtg.coxph', subdir="src")
```

This is the code to run the coxph:

```
print( client$getCollaborations() )

# Should output something like this:
#   id     name
# 1  1 ZEPPELIN
# 2  2 PIPELINE

# Select a collaboration
client$setCollaborationId(1)

# Define explanatory variables, time column and censor column
expl_vars <- c("Age","Race2","Race3","Mar2","Mar3","Mar4","Mar5","Mar9",
               "Hist8520","hist8522","hist8480","hist8501","hist8201",
               "hist8211","grade","ts","nne","npn","er2","er4")
time_col <- "Time"
censor_col <- "Censor"

# vtg.coxph contains the function `dcoxph`.
result <- vtg.coxph::dcoxph(client, expl_vars, time_col, censor_col)
```

4.2.4 API

The API can be called via HTTP requests from a programming language of your choice. You can explore how to use the server API on `https://<serverdomain>/apidocs` (e.g. https://petronas.vantage6.ai/apidocs for our Petronas server). This page will show you which API endpoints exist and how you can use them.

4.3 Node admin guide

This section shows you how you can set up your own vantage6 node. First, we discuss the requirements for your node machine, then guide you through the installation process. Finally, we explain how to configure and start your node.

4.3.1 Requirements

Note: This section is the same as the *server requirements* section - their requirements are very similar.

The (minimal) requirements of the node and server are similar. Note that these are recommendations: it may also work on other hardware, operating systems, versions of Python etc. (but they are not tested as much).

Hardware

- x86 CPU architecture + virtualization enabled

- 1 GB memory
- 50GB+ storage
- Stable and fast (1 Mbps+ internet connection)
- Public IP address

Software

- Operating system: - Ubuntu 18.04+ - MacOS Big Sur+ (only for node) - Windows 10+ (only for node)
- *Python*
- *Docker*

Note: For the server, Ubuntu is highly recommended. It is possible to run a development server (using *vserver start*) on Windows or MacOS, but for production purposes we recommend using Ubuntu.

Warning: The hardware requirements of the node also depend on the algorithms that the node will run. For example, you need much less compute power for a descriptive statistical algorithm than for a machine learning model.

4.3.1.1 Python

Installation of any of the vantage6 packages requires Python 3.10. For installation instructions, see python.org, anaconda.com or use the package manager native to your OS and/or distribution.

Note: We recommend you install vantage6 in a new, clean Python (Conda) environment.

Higher versions of Python (3.11+) will most likely also work, as might lower versions (3.8 or 3.9). However, we develop and test vantage6 on version 3.10, so that is the safest choice.

Warning: Note that Python 3.10 is only used in vantage6 versions 3.8.0 and higher. In lower versions, Python 3.7 is required.

4.3.1.2 Docker

Docker facilitates encapsulation of applications and their dependencies in packages that can be easily distributed to diverse systems. Algorithms are stored in Docker images which nodes can download and execute. Besides the algorithms, both the node and server are also running from a docker container themselves.

Please refer to this page on how to install Docker. To verify that Docker is installed and running you can run the `hello-world` example from Docker.

```
docker run hello-world
```

Warning: Note that for **Linux**, some post-installation steps may be required. Vantage6 needs to be able to run docker without `sudo`, and these steps ensure just that.

Note:

- Always make sure that Docker is running while using vantage6!
- We recommend to always use the latest version of Docker.

4.3.2 Install

To install the **vantage6-node** make sure you have met the requirements. Then, we provide a command-line interface (CLI) with which you can manage your node. The CLI is a Python package that can be installed using pip. We always recommend to install the CLI in a virtual environment or a conda environment.

Run this command to install the CLI in your environment:

```
pip install vantage6
```

Or if you want to install a specific version:

```
pip install vantage6==x.y.z
```

You can verify that the CLI has been installed by running the command `vnode --help`. If that prints a list of commands, the installation is completed.

The next pages will explain to configure, start and stop the node. The node software itself will be downloaded when you start the node for the first time.

4.3.3 Use

This section explains which commands are available to manage your node.

4.3.3.1 Quick start

To create a new node, run the command below. A menu will be started that allows you to set up a node configuration file. For more details, check out the *Configure* section.

```
vnode new
```

To run a node, execute the command below. The `--attach` flag will cause log output to be printed to the console.

```
vnode start --name <your_node> --attach
```

Finally, a node can be stopped again with:

```
vnode stop --name <your_node>
```

4.3.3.2 Available commands

Below is a list of all commands you can run for your node(s). To see all available options per command use the `--help` flag, i.e. `vnode start --help` .

Command	Description
`vnode new`	Create a new node configuration file
`vnode start`	Start a node
`vnode stop`	Stop a nodes
`vnode files`	List the files of a node (e.g. config and log files)
`vnode attach`	Print the node logs to the console
`vnode list`	List all existing nodes
`vnode create-private-key`	Create and upload a new public key for your organization
`vnode set-api-key`	Update the API key in your node configuration file

Local test setup

Check the section on *Local test setup* of the server if you want to run both the node and server on the same machine.

4.3.4 Configure

The vantage6-node requires a configuration file to run. This is a `yaml` file with a specific format.

The next sections describes how to configure the node. It first provides a few quick answers on setting up your node, then shows an example of all configuration file options, and finally explains where your vantage6 configuration files are stored.

4.3.4.1 How to create a configuration file

The easiest way to create an initial configuration file is via: `vnode new`. This allows you to configure the basic settings. For more advanced configuration options, which are listed below, you can view the *example configuration file*.

4.3.4.2 Where is my configuration file?

To see where your configuration file is located, you can use the following command

```
vnode files
```

Warning: This command will not work if you have put your configuration file in a custom location. Also, you may need to specify the `--system` flag if you put your configuration file in the *system folder*.

4.3.4.3 All configuration options

The following configuration file is an example that intends to list all possible configuration options.

You can download this file here: `node_config.yaml`

```yaml
application:
  # API key used to authenticate at the server.
  api_key: ***

  # URL of the vantage6 server
  server_url: https://petronas.vantage6.ai

  # port the server listens to
  port: 443

  # API path prefix that the server uses. Usually '/api' or an empty string
  api_path: ''

  # subnet of the VPN server
  vpn_subnet: 10.76.0.0/16

  # add additional environment variables to the algorithm containers.
  # this could be usefull for passwords or other things that algorithms
  # need to know about the node it is running on
  # OPTIONAL
  algorithm_env:

    # in this example the environment variable 'player' has
    # the value 'Alice' inside the algorithm container
    player: Alice

  # specify custom Docker images to use for starting the different
  # components.
  # OPTIONAL
  images:
    node: harbor2.vantage6.ai/infrastructure/node:petronas
    alpine: harbor2.vantage6.ai/infrastructure/alpine
    vpn_client: harbor2.vantage6.ai/infrastructure/vpn_client
    network_config: harbor2.vantage6.ai/infrastructure/vpn_network

  # path or endpoint to the local data source. The client can request a
  # certain database by using its label. The type is used by the
  # auto_wrapper method used by algorithms. This way the algorithm wrapper
  # knows how to read the data from the source. The auto_wrapper currently
  # supports: 'csv', 'parquet', 'sql', 'sparql', 'excel', 'omop'. If your
  # algorithm does not use the wrapper and you have a different type of
  # data source you can specify 'other'.
  databases:
    - label: default
      uri: D:\data\datafile.csv
      type: csv

  # end-to-end encryption settings
```

(continues on next page)

(continued from previous page)

```
encryption:

  # whenever encryption is enabled or not. This should be the same
  # as the `encrypted` setting of the collaboration to which this
  # node belongs.
  enabled: false

  # location to the private key file
  private_key: /path/to/private_key.pem

# Define who is allowed to run which algorithms on this node.
policies:
  # Control which algorithm images are allowed to run on this node. This is
  # expected to be a valid regular expression.
  allowed_algorithms:
    - ^harbor2.vantage6.ai/[a-zA-Z]+/[a-zA-Z]+
    - myalgorithm.ai/some-algorithm
  # Define which users are allowed to run algorithms on your node by their ID
  allowed_users:
    - 2
  # Define which organizations are allowed to run images on your node by
  # their ID or name
  allowed_organizations:
    - 6
    - root

# credentials used to login to private Docker registries
docker_registries:
  - registry: docker-registry.org
    username: docker-registry-user
    password: docker-registry-password

# Create SSH Tunnel to connect algorithms to external data sources. The
# `hostname` and `tunnel:bind:port` can be used by the algorithm
# container to connect to the external data source. This is the address
# you need to use in the `databases` section of the configuration file!
ssh-tunnels:

  # Hostname to be used within the internal network. I.e. this is the
  # hostname that the algorithm uses to connect to the data source. Make
  # sure this is unique and the same as what you specified in the
  # `databases` section of the configuration file.
  - hostname: my-data-source

    # SSH configuration of the remote machine
    ssh:

      # Hostname or ip of the remote machine, in case it is the docker
      # host you can use `host.docker.internal` for Windows and MacOS.
      # In the case of Linux you can use `172.17.0.1` (the ip of the
      # docker bridge on the host)
      host: host.docker.internal
```

(continues on next page)

(continued from previous page)

```
      port: 22

      # fingerprint of the remote machine. This is used to verify the
      # authenticity of the remote machine.
      fingerprint: "ssh-rsa ..."

      # Username and private key to use for authentication on the remote
      # machine
      identity:
        username: username
        key: /path/to/private_key.pem

      # Once the SSH connection is established, a tunnel is created to
      # forward traffic from the local machine to the remote machine.
      tunnel:

        # The port and ip on the tunnel container. The ip is always
        # 0.0.0.0 as we want the algorithm container to be able to
        # connect.
        bind:
          ip: 0.0.0.0
          port: 8000

        # The port and ip on the remote machine. If the data source runs
        # on this machine, the ip most likely is 127.0.0.1.
        dest:
          ip: 127.0.0.1
          port: 8000

# Settings for the logger
logging:
  # Controls the logging output level. Could be one of the following
  # levels: CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET
  level:        DEBUG

  # Filename of the log-file, used by RotatingFileHandler
  file:         my_node.log

  # whenever the output needs to be shown in the console
  use_console:  true

  # The number of log files that are kept, used by RotatingFileHandler
  backup_count: 5

  # Size kb of a single log file, used by RotatingFileHandler
  max_size:     1024

  # format: input for logging.Formatter,
  format:       "%(asctime)s - %(name)-14s - %(levelname)-8s - %(message)s"
  datefmt:      "%Y-%m-%d %H:%M:%S"

# directory where local task files (input/output) are stored
```

(continues on next page)

(continued from previous page)

```
task_dir: C:\Users\<your-user>\AppData\Local\vantage6\node\mydir

# Whether or not your node shares some configuration (e.g. which images are
# allowed to run on your node) with the central server. This can be useful
# for other organizations in your collaboration to understand why a task
# is not completed. Obviously, no sensitive data is shared. Default true
share_config: true
```

Note: We use DTAP for key environments. In short:

- `dev`: Development environment. It is ok to break things here
- `test`: Testing environment. Here, you can verify that everything works as expected. This environment should resemble the target environment where the final solution will be deployed as much as possible.
- `acc`: Acceptance environment. If the tests were successful, you can try this environment, where the final user will test his/her analysis to verify if everything meets his/her expectations.
- `prod`: Production environment. The version of the proposed solution where the final analyses are executed.

You can also specify the key `application` if you do not want to specify one of the environments. This is also done in the example configuration shown above.

4.3.4.4 Configuration file location

The directory where the configuration file is stored depends on your operating system (OS). It is possible to store the configuration file at **system** or at **user** level. By default, node configuration files are stored at **user** level, which makes this configuration available only for your user.

The default directories per OS are as follows:

Operating System	System-folder	User-folder
Windows	`C:\ProgramData\vantage\node\`	`C:\Users\<user>\AppData\Local\vantage\node\`
MacOS	`/Library/Application/Support/vantage6/node/`	`/Users/<user>/Library/Application Support/vantage6/node/`
Linux	`/etc/vantage6/node/`	`/home/<user>/.config/vantage6/node/`

Note: The command `vnode` looks in these directories by default. However, it is possible to use any directory and specify the location with the `--config` flag. But note that doing that requires you to specify the `--config` flag every time you execute a `vnode` command!

Similarly, you can put your node configuration file in the system folder by using the `--system` flag. Note that in that case, you have to specify the `--system` flag for all `vnode` commands.

4.3.4.5 Security

As a data owner it is important that you take the necessary steps to protect your data. Vantage6 allows algorithms to run on your data and share the results with other parties. It is important that you review the algorithms before allowing them to run on your data.

Once you approved the algorithm, it is important that you can verify that the approved algorithm is the algorithm that runs on your data. There are two important steps to be taken to accomplish this:

- Set the (optional) `allowed_images` option in the node-configuration file. You can specify a list of regex expressions here. Some examples of what you could define:
  1. `^harbor2.vantage6.ai/[a-zA-Z]+/[a-zA-Z]+`: allow all images from the vantage6 registry
  2. `^harbor2.vantage6.ai/algorithms/glm`: only allow the GLM image, but all builds of this image
  3. `^harbor2.vantage6.ai/algorithms/glm@sha256:82becede498899ec668628e7cb0ad87b6e1c371cb8a1e597d83a47fac21d6af3`: allows only this specific build from the GLM image to run on your data
- Enable `DOCKER_CONTENT_TRUST` to verify the origin of the image. For more details see the documentation from Docker.

Warning: By enabling `DOCKER_CONTENT_TRUST` you might not be able to use certain algorithms. You can check this by verifying that the images you want to be used are signed.

In case you are using our Docker repository you need to use harbor**2**.vantage6.ai as harbor.vantage6.ai does not have a notary.

4.3.4.6 Logging

To configure the logger, look at the logging section in the example configuration file in *All configuration options*.

Useful commands:

1. `vnode files`: shows you where the log file is stored
2. `vnode attach`: shows live logs of a running server in your current console. This can also be achieved when starting the node with `vnode start --attach`

4.4 Server admin guide

This section shows you how you can set up your own vantage6 server. First, we discuss the requirements for your server machine, then guide you through the installation process. Finally, we explain how to configure and start your server.

4.4.1 Requirements

Note: This section is the same as the *node requirements* section - their requirements are very similar.

The (minimal) requirements of the node and server are similar. Note that these are recommendations: it may also work on other hardware, operating systems, versions of Python etc. (but they are not tested as much).

Hardware

- x86 CPU architecture + virtualization enabled

- 1 GB memory
- 50GB+ storage
- Stable and fast (1 Mbps+ internet connection)
- Public IP address

Software

- Operating system: - Ubuntu 18.04+ - MacOS Big Sur+ (only for node) - Windows 10+ (only for node)
- *Python*
- *Docker*

Note: For the server, Ubuntu is highly recommended. It is possible to run a development server (using *vserver start*) on Windows or MacOS, but for production purposes we recommend using Ubuntu.

Warning: The hardware requirements of the node also depend on the algorithms that the node will run. For example, you need much less compute power for a descriptive statistical algorithm than for a machine learning model.

4.4.1.1 Python

Installation of any of the vantage6 packages requires Python 3.10. For installation instructions, see python.org, anaconda.com or use the package manager native to your OS and/or distribution.

Note: We recommend you install vantage6 in a new, clean Python (Conda) environment.

Higher versions of Python (3.11+) will most likely also work, as might lower versions (3.8 or 3.9). However, we develop and test vantage6 on version 3.10, so that is the safest choice.

Warning: Note that Python 3.10 is only used in vantage6 versions 3.8.0 and higher. In lower versions, Python 3.7 is required.

4.4.1.2 Docker

Docker facilitates encapsulation of applications and their dependencies in packages that can be easily distributed to diverse systems. Algorithms are stored in Docker images which nodes can download and execute. Besides the algorithms, both the node and server are also running from a docker container themselves.

Please refer to this page on how to install Docker. To verify that Docker is installed and running you can run the `hello-world` example from Docker.

```
docker run hello-world
```

Warning: Note that for **Linux**, some post-installation steps may be required. Vantage6 needs to be able to run docker without `sudo`, and these steps ensure just that.

Note:

- Always make sure that Docker is running while using vantage6!
- We recommend to always use the latest version of Docker.

4.4.2 Install

4.4.2.1 Local (test) Installation

To install the **vantage6 server**, make sure you have met the requirements. Then, we provide a command-line interface (CLI) with which you can manage your server. The CLI is a Python package that can be installed using pip. We always recommend to install the CLI in a virtual environment or a conda environment.

Run this command to install the CLI in your environment:

```
pip install vantage6
```

Or if you want to install a specific version:

```
pip install vantage6==x.y.z
```

You can verify that the CLI has been installed by running the command `vserver --help`. If that prints a list of commands, the installation is completed.

The server software itself will be downloaded when you start the server for the first time.

4.4.2.2 Host your server

To host your server, we recommend to use the Docker image we provide: `harbor2.vantage6.ai/infrastructure/server`. Running this docker image will start the server. Check the *Deploy* section for deployment examples.

Note: We recommend to use the latest version. Should you have reasons to deploy an older `VERSION`, use the image `harbor2.vantage6.ai/infrastructure/server:<VERSION>`.

If you deploy an older version, it is also recommended that the nodes match that version. They can do that by specifying the `--image` flag in their configuration file (see *this section* on node configuration).

4.4.3 Deploy

The vantage6 server is a Flask application, that uses python-socketio for socketIO connections. The server runs as a standalone process (listening on its own ip address/port).

There are many deployment options. We simply provide a few examples.

- *NGINX*
- *Azure app service*
- ...

Note: From version 3.2+ it is possible to horizontally scale the server (This upgrade is also made available to version 2.3.4)

Documentation on how to deploy it will be shared here soon. Reach out to us on Discord for now.

4.4.3.1 NGINX

A basic setup is shown below. Note that SSL is not configured in this example.

```
server {

    # Public port
    listen 80;
    server_name _;

    # vantage6-server. In the case you use a sub-path here, make sure
    # to foward also it to the proxy_pass
    location /subpath {
        include proxy_params;

        # internal ip and port
        proxy_pass http://127.0.0.1:5000/subpath;
    }

    # Allow the websocket traffic
    location /socket.io {
        include proxy_params;
        proxy_http_version 1.1;
        proxy_buffering off;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
        proxy_pass http://127.0.0.1:5000/socket.io;
    }
}
```

Note: When you *Configure* the server, make sure to include the `/subpath` that has been set in the NGINX configuration into the `api_path` setting (e.g. `api_path:  /subpath/api`)

4.4.3.2 Azure app service

Note: We still have to document this. Reach out to us on Discord for now.

4.4.4 Install optional components

There are several optional components that you can set up apart from the vantage6 server itself.

User interface
: An application that will allow your server's users to interact more easily with your vantage6 server.

EduVPN
: If you want to enable algorithm containers that are running on different nodes, to directly communicate with one another, you require a VPN server.

RabbitMQ
: If you have a server with a high workload whose performance you want to improve, you may want to set up a RabbitMQ service which enables horizontal scaling of the Vantage6 server.

Docker registry
: A (private) Docker registry can be used to store algorithms but it is also possible to use the (public) Docker hub to upload your Docker images.

SMTP server
: If you want to send emails to your users, e.g. to help them reset their password, you need to set up an SMTP server.

Below, we explain how to install and deploy these components.

4.4.4.1 User Interface

The User Interface (UI) is a web application that will make it easier for your users to interact with the server. It allows you to manage all your resources (such as creating collaborations, editing users, or viewing tasks), except for creating new tasks. We aim to incorporate this functionality in the near future.

To deploy a UI, follow the instructions on the UI Github page. We also provide a Docker image that runs the UI.

The UI is not compatible with older versions (<3.3) of vantage6.

4.4.4.2 EduVPN

EduVPN is an optional server component that enables the use of algorithms that require node-to-node communication.

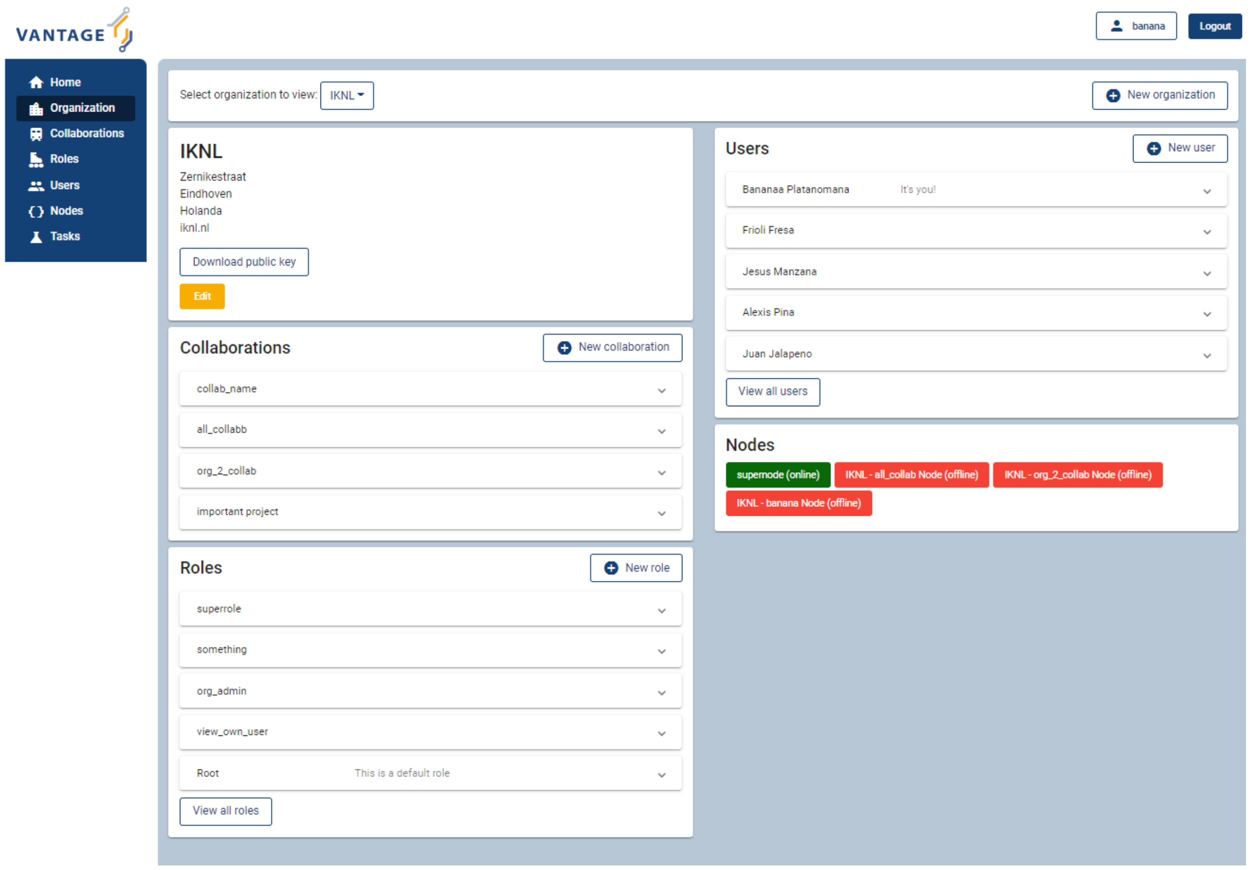
EduVPN provides an API for the OpenVPN server, which is required for automated certificate retrieval by the nodes. Like vantage6, it is an open source platform.

The following documentation shows you how to install EduVPN:

- Debian
- Centos
- Fedora

After the installation is done, you need to configure the server to:

1. Enable client-to-client communication. This can be achieved in the configuration file by the `clientToClient` setting (see here).
2. Do not block LAN communication (set `blockLan` to `false`). This allows your docker subnetworks to continue to communicate, which is required for vantage6 to function normally.
3. Enable port sharing (Optional). This may be useful if the nodes are behind a strict firewall. Port sharing allows nodes to connect to the VPN server only using outgoing `tcp/443`. Be aware that TCP meltdown can occur when using the TCP protocol for VPN.



Select organization to view: IKNL

IKNL
Zernikestraat
Eindhoven
Holanda
iknl.nl

Collaborations
- collab_name
- all_collabb
- org_2_collab
- important project

Roles
- superrole
- something
- org_admin
- view_own_user
- Root This is a default role

Users
- Bananaa Platanomana It's you!
- Frioli Fresa
- Jesus Manzana
- Alexis Pina
- Juan Jalapeno

Nodes
supernode (online) IKNL - all_collab Node (offline) IKNL - org_2_collab Node (offline) IKNL - banana Node (offline)

Fig. 4.5: Screenshot of the vantage6 UI

4. Create an application account.

Warning: EduVPN enables listening to multiple protocols (UDP/TCP) and ports at the same time. Be aware that all nodes need to be connected using the same protocol and port in order to communicate with each other.

Warning: The EduVPN server should usually be available to the public internet to allow all nodes to find it. Therefore, it should be properly secured, for example by closing all public ports (except http/https).

Additionally, you may want to explicitly allow *only* VPN traffic between nodes, and not between a node and the VPN server. You can achieve that by updating the firewall rules on your machine.

On Debian machines, these rules can be found in */etc/iptables/rules.v4* and */etc/iptables/rules.v6*, on CentOS, Red Hat Enterprise Linux and Fedora they can be found in */etc/sysconfig/iptables* and */etc/sysconfig/ip6tables*. You will have to do the following:

```
# In the firewall rules, below INPUT in the #SSH section, add this line
# to drop all VPN traffic with the VPN server as final destination:
-I INPUT -i tun+ -j DROP

# We only want to allow nodes to reach other nodes, and not other
# network interfaces available in the VPN.
# To achieve, replace the following rules:
-A FORWARD -i tun+ ! -o tun+ -j ACCEPT
-A FORWARD ! -i tun+ -o tun+ -j ACCEPT
# with:
-A FORWARD -i tun+ -o tun+ -j ACCEPT
-A FORWARD -i tun+ -j DROP
```

Example configuration

The following configuration makes a server listens to `TCP/443` only. Make sure you set `clientToClient` to `true` and `blockLan` to `false`. The `range` needs to be supplied to the node configuration files. Also note that the server configured below uses port-sharing.

```
// /etc/vpn-server-api/config.php
<?php

return [
    // List of VPN profiles
    'vpnProfiles' => [
        'internet' => [
            // The number of this profile, every profile per instance has a
            // unique number
            // REQUIRED
            'profileNumber' => 1,

            // The name of the profile as shown in the user and admin portals
            // REQUIRED
            'displayName' => 'vantage6 :: vpn service',

            // The IPv4 range of the network that will be assigned to clients
            // REQUIRED
```

(continues on next page)

(continued from previous page)

```
            'range' => '10.76.0.0/16',

            // The IPv6 range of the network that will be assigned to clients
            // REQUIRED
            'range6' => 'fd58:63db:3245:d20d::/64',

            // The hostname the VPN client(s) will connect to
            // REQUIRED
            'hostName' => 'eduvpn.vantage6.ai',

            // The address the OpenVPN processes will listen on
            // DEFAULT = '::'
            'listen' => '::',

            // The IP address to use for connecting to OpenVPN processes
            // DEFAULT = '127.0.0.1'
            'managementIp' => '127.0.0.1',

            // Whether or not to route all traffic from the client over the VPN
            // DEFAULT = false
            'defaultGateway' => true,

            // Block access to local LAN when VPN is active
            // DEFAULT = false
            'blockLan' => false,

            // IPv4 and IPv6 routes to push to the client, only used when
            // defaultGateway is false
            // DEFAULT = []
            'routes' => [],

            // IPv4 and IPv6 address of DNS server(s) to push to the client
            // DEFAULT  = []
            // Quad9 (https://www.quad9.net)
            'dns' => ['9.9.9.9', '2620:fe::fe'],

            // Whether or not to allow client-to-client traffic
            // DEFAULT = false
            'clientToClient' => true,

            // Whether or not to enable OpenVPN logging
            // DEFAULT = false
            'enableLog' => false,

            // Whether or not to enable ACLs for controlling who can connect
            // DEFAULT = false
            'enableAcl' => false,

            // The list of permissions to allow access, requires enableAcl to
            // be true
            // DEFAULT  = []
            'aclPermissionList' => [],
```

(continues on next page)

(continued from previous page)

```
            // The protocols and ports the OpenVPN processes should use, MUST
            // be either 1, 2, 4, 8 or 16 proto/port combinations
            // DEFAULT = ['udp/1194', 'tcp/1194']
            'vpnProtoPorts' => [
                'tcp/1195',
            ],

            // List the protocols and ports exposed to the VPN clients. Useful
            // for OpenVPN port sharing. When empty (or missing), uses list
            // from vpnProtoPorts
            // DEFAULT = []
            'exposedVpnProtoPorts' => [
                'tcp/443',
            ],

            // Hide the profile from the user portal, i.e. do not allow the
            // user to choose it
            // DEFAULT = false
            'hideProfile' => false,

            // Protect to TLS control channel with PSK
            // DEFAULT = tls-crypt
            'tlsProtection' => 'tls-crypt',
            //'tlsProtection' => false,
        ],
    ],

    // API consumers & credentials
    'apiConsumers' => [
        'vpn-user-portal' => '***',
        'vpn-server-node' => '***',
    ],
];
```

The following configuration snippet can be used to add an API user. The username and the `client_secret` have to be added to the vantage6-server configuration file.

```
...
'Api' => [
  'consumerList' => [
    'vantage6-user' => [
      'redirect_uri_list' => [
        'http://localhost',
      ],
      'display_name' => 'vantage6',
      'require_approval' => false,
      'client_secret' => '***'
    ]
  ]
...
```

4.4.4.3 RabbitMQ

RabbitMQ is an optional component that enables the server to handle more requests at the same time. This is important if a server has a high workload.

There are several options to host your own RabbitMQ server. You can run RabbitMQ in Docker or host RabbitMQ on Azure. When you have set up your RabbitMQ service, you can connect the server to it by adding the following to the server configuration:

```
rabbitmq_uri: amqp://<username>:<password@<hostname>:5672/<vhost>
```

Be sure to create the user and vhost that you specify exist! Otherwise, you can add them via the RabbitMQ management console.

4.4.4.4 Docker registry

A Docker registry or repository provides storage and versioning for Docker images. Installing a private Docker registry is useful if you want don't want to share your algorithms.

Docker Hub

Docker itself provides a registry as a turn-key solution on Docker Hub. Instructions for setting it up can be found here: https://hub.docker.com/_/registry.

Harbor

Harbor is another option for running a registry. Harbor provides access control, a user interface and automated scanning on vulnerabilities.

4.4.4.5 SMTP server

Some features of the server require an SMTP server to send emails. For example, the server can send an email to a user when they lost their password. There are many ways to set up an SMTP server, and we will not go into detail here. Just remember that you need to configure the server to use your SMTP server (see *All configuration options*).

4.4.5 Use

This section explains which commands are available to manage your server. It also explains how to set up a test server locally and how to manage resources via an IPython shell.

4.4.5.1 Quick start

To create a new server, run the command below. A menu will be started that allows you to set up a server configuration file.

```
vserver new
```

For more details, check out the *Configure* section.

To run a server, execute the command below. The `--attach` flag will copy log output to the console.

```
vserver start --name <your_server> --attach
```

Warning: When the server is run for the first time, the following user is created:

- username: root
- password: root

It is recommended to change this password immediately.

Finally, a server can be stopped again with:

```
vserver stop --name <your_server>
```

4.4.5.2 Available commands

The following commands are available in your environment. To see all the options that are available per command use the `--help` flag, e.g. `vserver start --help`.

Command	Description
`vserver new`	Create a new server configuration file
`vserver start`	Start a server
`vserver stop`	Stop a server
`vserver files`	List the files that a server is using
`vserver attach`	Show a server's logs in the current terminal
`vserver list`	List the available server instances
`vserver shell`	Run a server instance python shell
`vserver import`	Import server entities as a batch
`vserver version`	Shows the versions of all the components of the running server

4.4.5.3 Local test setup

If the nodes and the server run at the same machine, you have to make sure that the node can reach the server.

Windows and MacOS

Setting the server IP to `0.0.0.0` makes the server reachable at your localhost (this is also the case when the dockerized version is used). In order for the node to reach this server, set the `server_url` setting to `host.docker.internal`.

Warning: On the **M1** mac the local server might not be reachable from your nodes as `host.docker.internal` does not seem to refer to the host machine. Reach out to us on Discourse for a solution if you need this!

Linux

You should bind the server to `0.0.0.0`. In the node configuration files, you can then use `http://172.17.0.1`, assuming you use the default docker network settings.

4.4.5.4 Batch import

You can easily create a set of test users, organizations and collaborations by using a batch import. To do this, use the `vserver import /path/to/file.yaml` command. An example `yaml` file is provided below.

You can download this file `here`.

```
application: {}
  # you may also add your configuration here and leave environments empty

environments:
  # name of the environment (should be 'test', 'prod', 'acc' or 'dev')
  test:

    # Human readable description of the server instance. This is to help
    # your peers to identify the server
    description: Test

    # Should be prod, acc, test or dev. In case the type is set to test
    # the JWT-tokens expiration is set to 1 day (default is 6 hours). The
    # other types can be used in future releases of vantage6
    type: test

    # IP adress to which the server binds. In case you specify 0.0.0.0
    # the server listens on all interfaces
    ip: 0.0.0.0

    # Port to which the server binds
    port: 5000

    # API path prefix. (i.e. https://yourdomain.org/api_path/<endpoint>). In the
    # case you use a referse proxy and use a subpath, make sure to include it
    # here also.
    api_path: /api

    # The URI to the server database. This should be a valid SQLAlchemy URI,
    # e.g. for an Sqlite database: sqlite:///database-name.sqlite,
    # or Postgres: postgresql://username:password@172.17.0.1/database).
    uri: sqlite:///test.sqlite

    # This should be set to false in production as this allows to completely
    # wipe the database in a single command. Useful to set to true when
    # testing/developing.
    allow_drop_all: True

    # The secret key used to generate JWT authorization tokens. This should
    # be kept secret as others are able to generate access tokens if they
    # know this secret. This parameter is optional. In case it is not
    # provided in the configuration it is generated each time the server
    # starts. Thereby invalidating all previous distributed keys.
    # OPTIONAL
    jwt_secret_key: super-secret-key! # recommended but optional

    # Settings for the logger
```

(continues on next page)

(continued from previous page)

```
logging:

  # Controls the logging output level. Could be one of the following
  # levels: CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET
  level:        DEBUG

  # Filename of the log-file, used by RotatingFileHandler
  file:         test.log

  # Whether the output is shown in the console or not
  use_console:  True

  # The number of log files that are kept, used by RotatingFileHandler
  backup_count: 5

  # Size in kB of a single log file, used by RotatingFileHandler
  max_size:     1024

  # format: input for logging.Formatter,
  format:       "%(asctime)s - %(name)-14s - %(levelname)-8s - %(message)s"
  datefmt:      "%Y-%m-%d %H:%M:%S"

# Configure a smtp mail server for the server to use for administrative
# purposes. e.g. allowing users to reset their password.
# OPTIONAL
smtp:
  port: 587
  server: smtp.yourmailserver.com
  username: your-username
  password: super-secret-password

# Set an email address you want to direct your users to for support
# (defaults to the address you set above in the SMTP server or otherwise
# to support@vantage6.ai)
support_email: your-support@email.com

# set how long reset token provided via email are valid (default 1 hour)
email_token_validity_minutes: 60

# set how long tokens and refresh tokens are valid (default 6 and 48
# hours, respectively)
token_expires_hours: 6
refresh_token_expires_hours: 48

# If algorithm containers need direct communication between each other
# the server also requires a VPN server. (!) This must be a EduVPN
# instance as vantage6 makes use of their API (!)
# OPTIONAL
vpn_server:
    # the URL of your VPN server
    url: https://your-vpn-server.ext
```

(continues on next page)

(continued from previous page)

```
        # OATH2 settings, make sure these are the same as in the
        # configuration file of your EduVPN instance
        redirect_url: http://localhost
        client_id: your_VPN_client_user_name
        client_secret: your_VPN_client_user_password

        # Username and password to acccess the EduVPN portal
        portal_username: your_eduvpn_portal_user_name
        portal_userpass: your_eduvpn_portal_user_password

  prod: {}
  acc: {}
  dev: {}
```

Warning: All users that are imported using `vserver import` receive the superuser role. We are looking into ways to also be able to import roles. For more background info refer to this issue.

4.4.6 Configure

The vantage6-server requires a configuration file to run. This is a `yaml` file with a specific format.

The next sections describes how to configure the server. It first provides a few quick answers on setting up your server, then shows an example of all configuration file options, and finally explains where your vantage6 configuration files are stored.

4.4.6.1 How to create a configuration file

The easiest way to create an initial configuration file is via: `vserver new`. This allows you to configure the basic settings. For more advanced configuration options, which are listed below, you can view the *example configuration file*.

4.4.6.2 Where is my configuration file?

To see where your configuration file is located, you can use the following command

```
vserver files
```

Warning: This command will usually only work for local test deployments of the vantage6 server. If you have deployed the server on a remote server, this command will probably not work.

Also, note that on local deployments you may need to specify the `--user` flag if you put your configuration file in the *user folder*.

You can create and edit this file manually. To create an initial configuration file you can also use the configuration wizard: `vserver new`.

4.4.6.3 All configuration options

The following configuration file is an example that intends to list all possible configuration options.

You can download this file here: `server_config.yaml`

```yaml
application: {}
  # you may also add your configuration here and leave environments empty

environments:
  # name of the environment (should be 'test', 'prod', 'acc' or 'dev')
  test:

    # Human readable description of the server instance. This is to help
    # your peers to identify the server
    description: Test

    # Should be prod, acc, test or dev. In case the type is set to test
    # the JWT-tokens expiration is set to 1 day (default is 6 hours). The
    # other types can be used in future releases of vantage6
    type: test

    # IP adress to which the server binds. In case you specify 0.0.0.0
    # the server listens on all interfaces
    ip: 0.0.0.0

    # Port to which the server binds
    port: 5000

    # API path prefix. (i.e. https://yourdomain.org/api_path/<endpoint>). In the
    # case you use a referse proxy and use a subpath, make sure to include it
    # here also.
    api_path: /api

    # The URI to the server database. This should be a valid SQLAlchemy URI,
    # e.g. for an Sqlite database: sqlite:///database-name.sqlite,
    # or Postgres: postgresql://username:password@172.17.0.1/database).
    uri: sqlite:///test.sqlite

    # This should be set to false in production as this allows to completely
    # wipe the database in a single command. Useful to set to true when
    # testing/developing.
    allow_drop_all: True

    # The secret key used to generate JWT authorization tokens. This should
    # be kept secret as others are able to generate access tokens if they
    # know this secret. This parameter is optional. In case it is not
    # provided in the configuration it is generated each time the server
    # starts. Thereby invalidating all previous distributed keys.
    # OPTIONAL
    jwt_secret_key: super-secret-key! # recommended but optional

    # Settings for the logger
    logging:
```

(continues on next page)

(continued from previous page)

```
    # Controls the logging output level. Could be one of the following
    # levels: CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET
    level:        DEBUG

    # Filename of the log-file, used by RotatingFileHandler
    file:         test.log

    # Whether the output is shown in the console or not
    use_console:  True

    # The number of log files that are kept, used by RotatingFileHandler
    backup_count: 5

    # Size in kB of a single log file, used by RotatingFileHandler
    max_size:     1024

    # format: input for logging.Formatter,
    format:       "%(asctime)s - %(name)-14s - %(levelname)-8s - %(message)s"
    datefmt:      "%Y-%m-%d %H:%M:%S"

  # Configure a smtp mail server for the server to use for administrative
  # purposes. e.g. allowing users to reset their password.
  # OPTIONAL
  smtp:
    port: 587
    server: smtp.yourmailserver.com
    username: your-username
    password: super-secret-password

  # Set an email address you want to direct your users to for support
  # (defaults to the address you set above in the SMTP server or otherwise
  # to support@vantage6.ai)
  support_email: your-support@email.com

  # set how long reset token provided via email are valid (default 1 hour)
  email_token_validity_minutes: 60

  # set how long tokens and refresh tokens are valid (default 6 and 48
  # hours, respectively)
  token_expires_hours: 6
  refresh_token_expires_hours: 48

  # If algorithm containers need direct communication between each other
  # the server also requires a VPN server. (!) This must be a EduVPN
  # instance as vantage6 makes use of their API (!)
  # OPTIONAL
  vpn_server:
      # the URL of your VPN server
      url: https://your-vpn-server.ext

      # OATH2 settings, make sure these are the same as in the
```

(continues on next page)

(continued from previous page)

```
        # configuration file of your EduVPN instance
        redirect_url: http://localhost
        client_id: your_VPN_client_user_name
        client_secret: your_VPN_client_user_password

        # Username and password to acccess the EduVPN portal
        portal_username: your_eduvpn_portal_user_name
        portal_userpass: your_eduvpn_portal_user_password

prod: {}
acc: {}
dev: {}
```

Note: We use DTAP for key environments. In short:

- `dev` Development environment. It is ok to break things here
- `test` Testing environment. Here, you can verify that everything works as expected. This environment should resemble the target environment where the final solution will be deployed as much as possible.
- `acc` Acceptance environment. If the tests were successful, you can try this environment, where the final user will test his/her analysis to verify if everything meets his/her expectations.
- `prod` Production environment. The version of the proposed solution where the final analyses are executed.

You can also specify the key `application` if you do not want to specify one of the environments.

4.4.6.4 Configuration file location

The directory where to store the configuration file depends on you operating system (OS). It is possible to store the configuration file at **system** or at **user** level. At the user level, configuration files are only available for your user. By default, server configuration files are stored at **system** level.

The default directories per OS are as follows:

OS	System	User
Windows	`C:\ProgramData\vantage\server`	`C:\Users\<user>\AppData\Local\vantage\server`
MacOS	`/Library/Application/Support/vantage6/server`	`/Users/<user>/Library/Application Support/vantage6/server`
Linux	`/etc/xdg/vantage6/server/`	`/home/<user>/.config/vantage6/server/`

Warning: The command `vserver` looks in certain directories by default. It is possible to use any directory and specify the location with the `--config` flag. However, note that using a different directory requires you to specify the `--config` flag every time!

Similarly, you can put your server configuration file in the user folder by using the `--user` flag. Note that in that case, you have to specify the `--user` flag for all `vserver` commands.

4.4.6.5 Logging

Logging is enabled by default. To configure the logger, look at the `logging` section in the example configuration in *All configuration options*.

Useful commands:

1. `vserver files`: shows you where the log file is stored
2. `vserver attach`: show live logs of a running server in your current console. This can also be achieved when starting the server with `vserver start --attach`

4.4.7 Shell

Warning: Using the server shell is not recommended. The shell is outdated and superseded by other tools. The shell offers a server admin the ability to manage the server entities, but does not offer any validation of the input. Therefore, it is easy to break the server by using the shell.

Instead, we recommend using the *user interface*, the *Python client* or the *API*.

The shell allows a server admin to manage all server entities. To start the shell, use `vserver shell [options]`.

In the next sections the different database models that are available are explained. You can retrieve any record and edit any property of it. Every `db.` object has a `help()` method which prints some info on what data is stored in it (e.g. `db.Organization.help()`).

Note: Don't forget to call `.save()` once you are done editing an object.

4.4.7.1 Organizations

Note: Organizations have a public key that is used for end-to-end encryption. This key is automatically created and/or uploaded by the node the first time it runs.

To store an organization you can use the `db.Organization` model:

```python
# create new organiztion
organization = db.Organization(
    name="IKNL",
    domain="iknl.nl",
    address1="Zernikestraat 29",
    address2="Eindhoven",
    zipcode="5612HZ",
    country="Netherlands"
)

# store organization in the database
organization.save()
```

Retrieving organizations from the database:

```
# get all organizations in the database
organizations = db.Organization.get()

# get organization by its unique id
organization = db.Organization.get(1)

# get organization by its name
organization = db.Organization.get_by_name("IKNL")
```

A lot of entities (e.g. users) at the server are connected to an organization. E.g. you can see which (computation) tasks are issued by the organization or see which collaborations it is participating in.

```
# retrieve organization from which we want to know more
organization = db.Organization.get_by_name("IKNL")

# get all collaborations in which the organization participates
collaborations = organization.collaborations

# get all users from the organization
users = organization.users

# get all created tasks (from all users)
tasks = organization.created_tasks

# get the results of all these tasks
results = organization.results

# get all nodes of this organization (for each collaboration
# an organization participates in, it needs a node)
nodes = organization.nodes
```

4.4.7.2 Roles and Rules

A user can have multiple roles and rules assigned to them. These are used to determine if the user has permission to view, edit, create or delete certain resources using the API. A role is a collection of rules.

```
# display all available rules
db.Rule.get()

# display rule 1
db.Rule.get(1)

# display all available roles
db.Role.get()

# display role 3
db.Role.get(3)

# show all rules that belong to role 3
db.Role.get(3).rules

# retrieve a certain rule from the DB
```

(continues on next page)

(continued from previous page)

```
rule = db.Rule.get_by_("node", Scope, Operation)

# create a new role
role = db.Role(name="role-name", rules=[rule])
role.save()

# or assign the rule directly to the user
user = db.User.get_by_username("some-existing-username")
user.rules.append(rule)
user.save()
```

4.4.7.3 Users

Users belong to an organization. So if you have not created any *Organizations* yet, then you should do that first. To create a user you can use the `db.User` model:

```
# first obtain the organization to which the new user belongs
org = db.Organization.get_by_name("IKNL")

# obtain role 3 to assign to the new user
role_3 = db.Role.get(3)

# create the new users, see section Roles and Rules on how to
# deal with permissions
new_user = db.User(
    username="root",
    password="super-secret",
    firstname="John",
    lastname="Doe",
    roles=[role_3],
    rules=[],
    organization=org
)

# store the user in the database
new_user.save()
```

You can retrieve users in the following ways:

```
# get all users
db.User.get()

# get user 1
db.User.get(1)

# get user by username
db.User.get_by_username("root")

# get all users from the organization IKNL
db.Organization.get_by_name("IKNL").users
```

To modify a user, simply adjust the properties and save the object.

```
user = db.User.get_by_username("some-existing-username")

# update the firstname
user.firstname = "Brandnew"

# update the password; it is automatically hashed.
user.password = "something-new"

# store the updated user in the database
user.save()
```

4.4.7.4 Collaborations

A collaboration consists of one or more organizations. To create a collaboration you need at least one but preferably multiple *Organizations* in your database. To create a collaboration you can use the `db.Collaboration` model:

```
# create a second organization to collaborate with
other_organization = db.Organization(
    name="IKNL",
    domain="iknl.nl",
    address1="Zernikestraat 29",
    address2="Eindhoven",
    zipcode="5612HZ",
    country="Netherlands"
)
other_organization.save()

# get organization we have created earlier
iknl = db.Organization.get_by_name("IKNL")

# create the collaboration
collaboration = db.Collaboration(
    name="collaboration-name",
    encrypted=False,
    organizations=[iknl, other_organization]
)

# store the collaboration in the database
collaboration.save()
```

Tasks, nodes and organizations are directly related to collaborations. We can obtain these by:

```
# obtain a collaboration which we like to inspect
collaboration = db.Collaboration.get(1)

# get all nodes
collaboration.nodes

# get all tasks issued for this collaboration
collaboration.tasks

```

(continues on next page)

(continued from previous page)

```python
# get all organizations
collaboration.organizations
```

Warning: Setting the encryption to False at the server does not mean that the nodes will send encrypted results. This is only the case if the nodes also agree on this setting. If they don't, you will receive an error message.

4.4.7.5 Nodes

Before nodes can login, they need to exist in the server's database. A new node can be created as follows:

```python
# we'll use a uuid as the API-key, but you can use anything as
# API key
from uuid import uuid1

# nodes always belong to an organization *and* a collaboration,
# this combination needs to be unique!
iknl = db.Organization.get_by_name("IKNL")
collab = iknl.collaborations[0]

# generate and save
api_key = str(uuid1())
print(api_key)

node = db.Node(
    name = f"IKNL Node - Collaboration {collab.name}",
    organization = iknl,
    collaboration = collab,
    api_key = api_key
)

# save the new node to the database
node.save()
```

Note: API keys are hashed before stored in the database. Therefore you need to save the API key immediately. If you lose it, you can reset the API key later via the shell, API, client or UI.

4.4.7.6 Tasks and Results

Warning: Tasks(/results) created from the shell are not picked up by nodes that are already running. The signal to notify them of a new task cannot be emitted this way. We therefore recommend sending tasks via the Python client.

A task is intended for one or more organizations. For each organization the task is intended for, a corresponding (initially empty) result should be created. Each task can have multiple results, for example a result from each organization.

```
# obtain organization from which this task is posted
iknl = db.Organization.get_by_name("IKNL")

# obtain collaboration for which we want to create a task
collaboration = db.Collaboration.get(1)

# obtain the next run_id. Tasks sharing the same run_id
# can share the temporary volumes at the nodes. Usually this
# run_id is assigned through the API (as the user is not allowed
# to do so). All tasks from a master-container share the
# same run_id
run_id = db.Task.next_run_id()

task = db.Task(
    name="some-name",
    description="some human readable description",
    image="docker-registry.org/image-name",
    collaboration=collaboration,
    run_id=run_id,
    database="default",
    initiator=iknl,
)
task.save()

# input the algorithm container (docker-registry.org/image-name)
# expects
input_ = {
}

import datetime

# now create a result model for each organization within the
# collaboration. This could also be a subset
for org in collaboration.organizations:
    res = db.Result(
        input=input_,
        organization=org,
        task=task,
        assigned_at=datetime.datetime.now()
    )
    res.save()
```

Tasks can have a child/parent relationship. Note that the `run_id` is for parent and child tasks the same.

```
# get a task to which we want to create some
# child tasks
parent_task = db.Task.get(1)

child_task = db.Task(
    name="some-name",
    description="some human readable description",
    image="docker-registry.org/image-name",
    collaboration=collaboration,
```

(continues on next page)

(continued from previous page)

```
    run_id=parent_task.run_id,
    database="default",
    initiator=iknl,
    parent=parent_task
)
child_task.save()
```

Note: Tasks that share a `run_id` have access to the same temporary folder at the node. This allows for multi-stage algorithms.

Obtaining results:

```
# obtain all Results
db.Result.get()

# obtain only completed results
[result for result in db.Result.get() if result.complete]

# obtain result by its unique id
db.Result.get(1)
```

4.5 Algorithm Development

This section helps you to develop MPC and FL algorithms that are compatible with vantage6. You are **not** going to find a list of algorithms here or help on how to use them. In the *Components* the basic concepts and interface between node and algorithm is explained. Then in the *Classic Tutorial* a FL algorithm is build from scratch.

This section is to be extended with more examples in the future.

4.5.1 Concepts

Algorithms are executed at the (vantage6-)node. The node receives a computation task from the vantage6-server. The node will then retrieve the algorithm, execute it and return the results to the server.

Algorithms are shared using Docker images which are stored in a Docker image registry which is accessible to the nodes. In the following sections we explain the fundamentals of algorithm containers.

1. *Input & output* Interface between the node and algorithm container
2. Wrappers Library to simplify and standardized the node-algorithm input and output
3. *Child containers* Creating subtasks from an algorithm container
4. *Networking* Communicate with other algorithm containers and the vantage6-server
5. *Cross language* Cross language data serialization
6. *Package & distribute* Packaging and shipping algorithms

4.5.1.1 Input & output

The algorithm runs in an isolated environment within the data station (node). As it is important to limit the connectivity and accessability for obvious security reasons. In order for the algorithm to do its work, it is provided with several resources.

Note: This section describes the current process. Keep in mind that this is subjected to be changed. For more information, please see this Github

Environment variables

The algorithms have access to several environment variables, see Section 4.5.1.1. These can be used to locate certain files or to add local configuration settings into the container.

Table 4.1: Environment variables

Variable	Description
`INPUT_FILE`	path to the input file. The input file contains the user defined input for the algorithms.
`TOKEN_FILE`	Path to the token file. The token file contains a JWT token which can be used to access the vantage6-server. This way the algorithm container is able to post new tasks and retrieve results.
`TEMPORARY_FOLDER`	Path to the temporary folder. This folder can be used to store intermediate results. These intermediate results are shared between all containers that have the same run_id. Algorithm containers which are created from an algorithm container themselves share the same run_id.
`HOST`	Contains the URL to the vantage6-server.
`PORT`	Contains the port to which the vantage6-server listens. Is used in combination with HOST and API_PATH.
`API_PATH`	Contains the api base path from the vantage6-server.
`[*]_DATABASE_URI`	Contains the URI of the local database. The * is replaced by the key specified in the node configuration file.

Note: Additional environment variables can be specified in the node configuration file using the algorithm_env key. These additional variables are forwarded to all algorithm containers.

File mounts

The algorithm container has access to several file mounts.

Input
: The input file contains the user defined input. The user specifies this when a task is created.

Output
: The algorithm should write its output to this file. When the docker container exits the contents of this file will be send back to the vantage6-server.

Token
: The token file contains a JWT token which can be used by the algorithm to communicate with the central server.

The token can only be used to create a new task with the same image, and is only valid while the task has not yet been completed.

Temporary directory

The temporary directory can be used by an algorithm container to share files with other algorithm containers that:

- run on the same node
- have the same `run_id`

Algorithm containers that origin from another container (a.k.a master container or parent container) share the same `run_id`. i.o. if a user creates a task a new `run_id` is assigned.

The paths to these files and directories are stored in the environment variables, which we will explain now.

4.5.1.2 Wrappers

The algorithm wrapper simplifies and standardizes the interaction between algorithm and node. The client libraries and the algorithm wrapper are tied together and use the same standards. The algorithm wrapper:

- reads the environment variables and file mounts and supplies these to your algorithm.
- provides an entrypoint for the docker container
- allows to write a single algorithm for multiple types of data sources

The wrapper is language specific and currently we support Python and R. Extending this concept to other languages is not so complex.

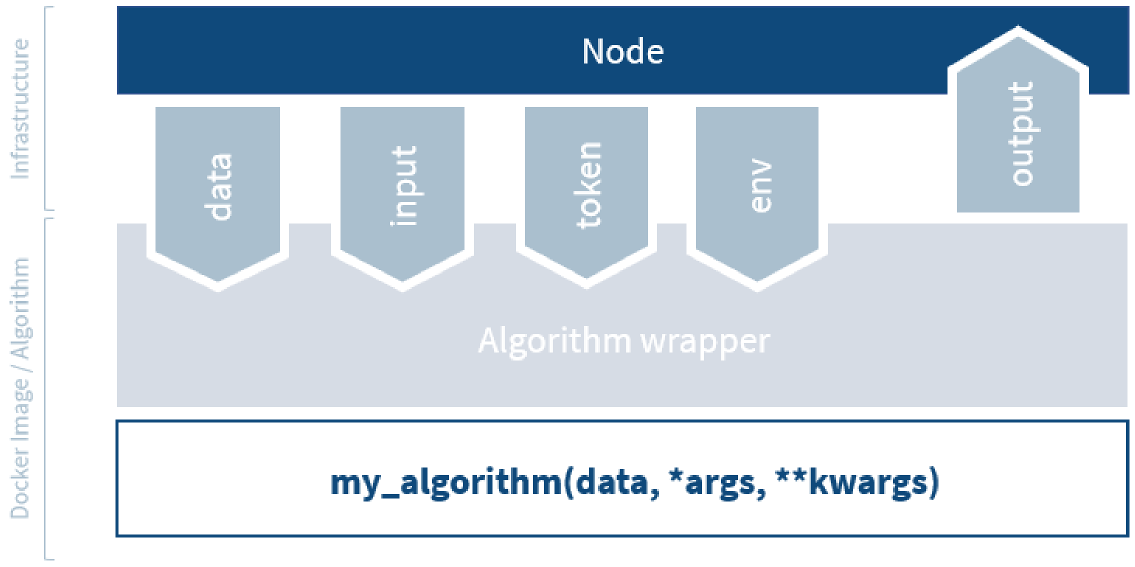


Fig. 4.6: The algorithm wrapper handles algorithm input and output.

Federated function

The signature of your function has to contain `data` as the first argument. The method name should have a `RPC_` prefix. Everything that is returned by the function will be written to the output file.

Python:

```python
def RPC_my_algorithm(data, *args, **kwargs):
    pass
```

R:

```r
RPC_my_algorithm <- function(data, ...) {
}
```

Central function

It is quite common to have a central part of your federated analysis which orchestrates the algorithm and combines the partial results. A common pattern for a central function would be:

1. Request partial models from all participants
2. Obtain the partial models
3. Combine the partial models to a global model
4. (optional) Repeat step 1-3 until the model converges

It is possible to run the central part of the analysis on your own machine, but it is also possible to let vantage6 handle the central part. There are several advantages to letting vantage6 handle this:

- You don't have to keep your machine running during the analysis
- You don't need to use the same programming language as the algorithm in case a language specific serialization is used in the algorithm

Note: Central functions also run at a node and *not* at the server.

In contrast to the federated functions, central functions are not prefixed. The first argument needs to be `client` and the second argument needs to be `data`. The `data` argument contains the local data and the `client` argument provides an interface to the vantage6-server.

Warning: The argument data is not present in the R wrapper. This is a consistency issue which will be solved in a future release.

```python
def main(client, data, *args, **kwargs):
   # Run a federated function. Note that we omnit the
   # RPC_ prefix. This prefix is added automatically
   # by the infrastructure
   task = client.create_new_task(
      {
         "method": "my_algorithm",
         "args": [],
```

(continues on next page)

(continued from previous page)

```
            "kwargs": {}
        },
        organization_ids=[...]
    )

    # wait for the federated part to complete
    # and return
    results = wait_and_collect(task)

    return results
```

```
main <- function(client, ...) {
    # Run a federated function. Note that we omnit the
    # RPC_ prefix. This prefix is added automatically
    # by the infrastructure
    result <- client$call("my_algorithm", ...)

    # Optionally do something with the results

    # return the results
    return(result)
}
```

Different wrappers

The docker wrappers read the local data source and supplies this to your functions in your algorithm. Currently CSV and SPARQL for Python and a CSV wrapper for R is supported. Since the wrapper handles the reading of the data, you need to rebuild your algorithm with a different wrapper to make it compatible with a different type of data source. You do this by updating the `CMD` directive in the dockerfile.

CSV wrapper (Python)

```
...
CMD python -c "from vantage6.tools.docker_wrapper import docker_wrapper; docker_wrapper('
↪${PKG_NAME}')"
```

CSV wrapper (R)

```
...
CMD Rscript -e "vtg::docker.wrapper('$PKG_NAME')"
```

SPARQL wrapper (Python)

```
...
CMD python -c "from vantage6.tools.docker_wrapper import sparql_wrapper; sparql_wrapper('
↪${PKG_NAME}')"
```

Parquet wrapper (Python)

```
...
CMD python -c "from vantage6.tools.docker_wrapper import parquet_wrapper; parquet_
↪wrapper('${PKG_NAME}')"
```

Data serialization

TODO

4.5.1.3 Mock client

TODO

4.5.1.4 Child containers

When a user creates a task, one or more nodes spawn an algorithm container. These algorithm containers can create new tasks themselves.

Every algorithm is supplied with a JWT token (see *Input & output*). This token can be used to communicate with the vantage6-server. In case you use a algorithm wrapper, you simply can use the supplied `Client` in case you use a *Central function*.

A child container can be a parent container itself. There is no limit to the amount of task layers that can be created. It is common to have only a single parent container which handles many child containers.

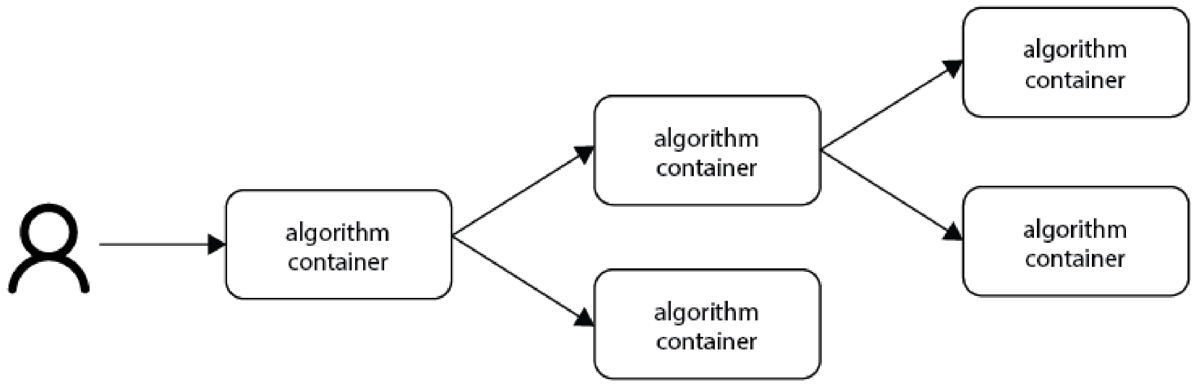


Fig. 4.7: Each container can spawn new containers in the network. Each container is provided with a unique token which they can use to communicate to the vantage6-server.

The token to which the containers have access supplies limited permissions to the container. For example, the token can be used to create additional tasks, but only in the same collaboration, and using the same image.

4.5.1.5 Networking

The algorithm container is deployed in an isolated network to reduce their exposure. Hence, the algorithm it cannot reach the internet. There are two exceptions:

1. When the VPN feature is enabled on the server all algorithm containers are able to reach each other using an `ip` and `port` over VPN.
2. The central server is reachable through a local proxy service. In the algorithm you can use the `HOST`, `POST` and `API_PATH` to find the address of the server.

Note: We are working on a whitelisting feature which allows a node to configure addresses that the algorithm container is able to reach.

VPN connection

Algorithm containers can expose one or more ports. These ports can then be used by other algorithm containers to exchange data. The infrastructure uses the Dockerfile from which the algorithm has been build to determine to which ports are used by the algorithm. This is done by using the `EXPOSE` and `LABEL` directives.

For example when an algorithm uses two ports, one port for communication `com` and one port for data exchange `data`. The following block should be added to you algorithm Dockerfile:

```
# port 8888 is used by the algorithm for communication purposes
EXPOSE 8888
LABEL p8888 = "com"

# port 8889 is used by the algorithm for data-exchange
EXPOSE 8889
LABEL p8889 = "data"
```

Port `8888` and `8889` are the internal ports to which the algorithm container listens. When another container want to communicate with this container it can retrieve the IP and external port from the central server by using the `result_id` and the label of the port you want to use (`com` or `data` in this case)

4.5.1.6 Cross language

Because algorithms are exchanged through Docker images they can be written in any language. This is an advantage as developers can use their preferred language for the problem they need to solve.

Warning: The wrappers are only available for R and Python, so when you use different language you need to handle the IO yourself. Consult the *Input & Output* section on what the node supplies to your algorithm container.

When data is exchanged between the user and the algorithm they both need to be able to read the data. When the algorithm uses a language specific serialization (e.g. a `pickle` in the case of Python or `RData` in the case of R) the user needs to use the same language to read the results. A better solution would be to use a type of serialization that is not specific to a language. For our wrappers we use JSON for this purpose.

Note: Communication between algorithm containers can use language specific serialization as long as the different parts of the algorithm use the same language.

4.5.1.7 Package & distribute

Once the algorithm is completed it needs to be packaged and made available for retrieval by the nodes. The algorithm is packaged in a Docker image. A Docker image is created from a Dockerfile, which acts as blue-print. Once the Docker image is created it needs to be uploaded to a registry so that nodes can retrieve it.

Dockerfile

A minimal Dockerfile should include a base image, injecting your algorithm and execution command of your algorithm. Here are several examples:

```
# python3 image as base
FROM python:3

# copy your algorithm in the container
COPY . /app

# maybe your algorithm is installable.
RUN pip install /app

# execute your application
CMD python /app/app.py
```

When using the Python Wrappers, the Dockerfile needs to follow a certain format. You should only change the `PKG_NAME` value to the Python package name of your algorithm.

```
# python vantage6 algorithm base image
FROM harbor.vantage6.ai/algorithms/algorithm-base

# this should reflect the python package name
ARG PKG_NAME="v6-summary-py"

# install federated algorithm
COPY . /app
RUN pip install /app

ENV PKG_NAME=${PKG_NAME}

# Tell docker to execute `docker_wrapper()` when the image is run.
CMD python -c "from vantage6.tools.docker_wrapper import docker_wrapper; docker_wrapper('
↪${PKG_NAME}'
```

Note: When using the python wrapper your algorithm file needs to be installable. See here for more information on how to create a python package.

When using the R Wrappers, the Dockerfile needs to follow a certain format. You should only change the `PKG_NAME` value to the R package name of your algorithm.

```
# The Dockerfile tells Docker how to construct the image with your algorithm.
# Once pushed to a repository, images can be downloaded and executed by the
# network hubs.
```

(continues on next page)

(continued from previous page)

```
FROM harbor2.vantage6.ai/base/custom-r-base

# this should reflect the R package name
ARG PKG_NAME='vtg.package'

LABEL maintainer="Main Tainer <m.tainer@vantage6.ai>"

# Install federated glm package
COPY . /usr/local/R/${PKG_NAME}/

WORKDIR /usr/local/R/${PKG_NAME}
RUN Rscript -e 'library(devtools)' -e 'install_deps(".")'
RUN R CMD INSTALL --no-multiarch --with-keep.source .

# Tell docker to execute `docker.wrapper()` when the image is run.
ENV PKG_NAME=${PKG_NAME}
CMD Rscript -e "vtg::docker.wrapper('$PKG_NAME')"
```

Note: Additional Docker directives are needed when using direct communication between different algorithm containers, see *Networking*.

Build & upload

If you are in the folder containing the Dockerfile, you can build the project as follows:

```
docker build -t repo/image:tag .
```

The `-t` indicated the name of your image. This name is also used as reference where the image is located on the internet. If you use Docker hub to store your images, you only specify your username as `repo` followed by your image name and tag: `USERNAME/IMAGE_NAME:IMAGE_TAG`. When using a private registry `repo` should contain the URL of the registry also: e.g. `harbor2.vantage6.ai/PROJECT/IMAGE_NAME:TAG`.

Then you can push you image:

```
docker push repo/image:tag
```

Now that is has been uploaded it is available for nodes to retrieve when they need it.

Signed images

It is possible to use the Docker the framework to create signed images. When using signed images, the node can verify the author of the algorithm image adding an additional protection layer.

Dockerfile

- Build project
- CMD
- Expose

4.5.2 Classic Tutorial

In this section the basic steps for creating an algorithm for horizontally partitioned data are explained.

Note: The final code of this tutorial is published on Github. The algorithm is also published in our Docker registry: *harbor2.vantage6.ai/demo/average*

It is assumed that it is mathematically possible to create a federated version of the algorithm you want to use. In the following sections we create a federated algorithm to compute the average of a distributed dataset. An overview of the steps that we are going through:

1. Mathematically decompose the model
2. Federated implementation and local testing
3. Vantage6 algorithm wrapper
4. Dockerize and push to a registry

This tutorial shows you how to create a **federated mean** algorithm.

4.5.2.1 Mathematical decomposition

The mean of $Q = [q_1, q_2 ... q_n]$ is computed as:

$$Q_{mean} = \frac{1}{n}\sum_{i=1}^{n} q_i = \frac{q_1 + q_2 + ... + q_n}{n}$$

When dataset Q is **horizontally partitioned** in dataset A and B:

$$A = [a_1, a_2 ... a_j] = [q_1, q_2 ... q_j]$$
$$B = [b_1, b_2 ... b_k] = [q_{j+1}, q_{j+2} ... q_n]$$

We would like to compute Q_{mean} from dataset A and B. This could be computed as:

$$Q_{mean} = \frac{(a_1 + a_2 + ... + a_j) + (b_1 + b_2 + ... + b_k)}{j + k} = \frac{\sum A + \sum B}{j + k}$$

Both the number of samples in each dataset and the total sum of each dataset is needed. Then we can compute the global average of dataset A and B.

Note: We cannot simply compute the average on each node and combine them, as this would be mathematically incorrect. This would only work if dataset **A** and **B** contain the exact same number of samples.

4.5.2.2 Federated implementation

Warning: In this example we use python, however you are free to use any language. The only requirements are: 1) It has to be able to create HTTP-requests, and 2) has to be able to read and write to files.

However, if you use a different language you are not able to use our wrapper. Reach out to us on Discord to discuss how this works.

A federated algorithm consist of two parts:

1. A federated part of the algorithm which is responsible for creating the partial results. In our case this would be computing (1) the sum of the observations, and (2) the number of observations.
2. A central part of the algorithm which is responsible for combining the partial results from the nodes. In the case of the federated mean that would be dividing the total sum of the observations by the total number of observations.

Note: The central part of the algorithm can either be run on the machine of the researcher himself or in a master container which runs on a node. The latter is the preferred method.

In case the researcher runs this part, he/she needs to have a proper setup to do so (i.e. Python 3.5+ and the necessary dependencies). This can be useful when developing new algorithms.

Federated part

The node that runs this part contains a CSV-file with one column (specified by the argument *column_name*) which we want to use to compute the global mean. We assume that this column has no *NaN* values.

```python
import pandas

def federated_part(path, column_name="numbers"):
    """Compute the sum and number of observations of a column"""

    # extract the column numbers from the CSV
    numbers = pandas.read_csv(path)[column_name]

    # compute the sum, and count number of rows
    local_sum = numbers.sum()
    local_count = len(numbers)

    # return the values as a dict
    return {
        "sum": local_sum,
        "count": local_count
    }
```

Central part

The central algorithm receives the sums and counts from all sites and combines these to a global mean. This could be from one or more sites.

```python
def central_part(node_outputs):
    """Combine the partial results to a global average"""
    global_sum = 0
    global_count = 0
    for output in node_outputs:
        global_sum += output["sum"]
        global_count += output["count"]

    return {"average": global_sum / global_count}
```

Local testing

To test, simply create two datasets **A** and **B**, both having a numerical column **numbers**. Then run the following:

```
outputs = [
    federated_part("path/to/dataset/A"),
    federated_part("path/to/dataset/B")
]
Q_average = central_part(outputs)["average"]
print(f"global average = {Q_average}.")
```

4.5.2.3 Vantage6 integration

Note: A good starting point would be to use the boilerplate code from our Github. This section outlines the steps needed to get to this boilerplate but also provides some background information.

Note: In this example we use a **csv**-file. It is also possible to use other types of data sources. This tutorial makes use of our algorithm wrapper which is currently only available for **csv**, **SPARQL** and **Parquet** files.

Other wrappers like **SQL**, **OMOP**, etc. are under consideration. Let us now if you want to use one of these or other datasources.

Now that we have a federated implementation of our algorithm we need to make it compatible with the vantage6 infrastructure. The infrastructure handles the communication with the server and provides data access to the algorithm.

The algorithm consumes a file containing the input. This contains both the method name to be triggered as well as the arguments provided to the method. The algorithm also has access to a CSV file (in the future this could also be a database) on which the algorithm can run. When the algorithm is finished, it writes back the output to a different file.

The central part of the algorithm has to be able to create (sub)tasks. These subtasks are responsible for executing the federated part of the algorithm. The central part of the algorithm can either be executed on one of the nodes in the vantage6 network or on the machine of a researcher. In this example we only show the case in which one of the nodes executes the central part of the algorithm. The node provides the algorithm with a JWT token so that the central part of the algorithm has access to the server to post these subtasks.

Algorithm Structure

The algorithm needs to be structured as a Python package. This way the algorithm can be installed within the Docker image. The minimal file-structure would be:

```
project_folder
├── Dockerfile
├── setup.py
└── algorithm_pkg
    └── __init__.py
```

We also recommend adding a `README.md`, `LICENSE` and `requirements.txt` to the *project_folder*.

setup.py

Contains the setup method to create a package from your algorithm code. Here you specify some details about your package and the dependencies it requires.

```python
from os import path
from codecs import open
from setuptools import setup, find_packages

# we're using a README.md, if you do not have this in your folder, simply
# replace this with a string.
here = path.abspath(path.dirname(__file__))
with open(path.join(here, 'README.md'), encoding='utf-8') as f:
    long_description = f.read()

# Here you specify the meta-data of your package. The `name` argument is
# needed in some other steps.
setup(
    name='v6-average-py',
    version="1.0.0",
    description='vantage6 average',
    long_description=long_description,
    long_description_content_type='text/markdown',
    url='https://github.com/IKNL/v6-average-py',
    packages=find_packages(),
    python_requires='>=3.6',
    install_requires=[
        'vantage6-client',
        # list your dependencies here:
        # pandas, ...
    ]
)
```

Note: The `setup.py` above is sufficient in most cases. However if you want to do more advanced stuff (like adding static data, or a CLI) you can use the extra arguments from `setup`.

Dockerfile

The Dockerfile contains the recipe for building the Docker image. Typically you only have to change the argument `PKG_NAME` to the name of you package. This name should be the same as as the name you specified in the `setup.py`. In our case that would be `v6-average-py`.

```
# This specifies our base image. This base image contains some commonly used
# dependancies and an install from all vantage6 packages. You can specify a
# different image here (e.g. python:3). In that case it is important that
# `vantage6-client` is a dependancy of you project as this contains the wrapper
# we are using in this example.
FROM harbor.vantage6.ai/algorithms/algorithm-base

# Change this to the package name of your project. This needs to be the same
```

(continues on next page)

(continued from previous page)

```
# as what you specified for the name in the `setup.py`.
ARG PKG_NAME="v6-average-py"

# This will install your algorithm into this image.
COPY . /app
RUN pip install /app

# This will run your algorithm when the Docker container is started. The
# wrapper takes care of the IO handling (communication between node and
# algorithm). You dont need to change anything here.
ENV PKG_NAME=${PKG_NAME}
CMD python -c "from vantage6.tools.docker_wrapper import docker_wrapper; docker_wrapper('
↪${PKG_NAME}')"
```

`__init__.py`

This contains the code for your algorithm. It is possible to split this into multiple files, however the methods that should be available to the researcher should be in this file. You can do that by simply importing them into this file (e.g. `from .average import my_nested_method`)

We can distinguish two types of methods that a user can trigger:

name	description	prefix	arguments
master	Central part of the algorithm. Receives a `client` as argument which provides an interface to the central server. This way the master can create tasks and collect their results.		`(client, data, *args, **kwargs)`
Remote procedure call	Consumes the data at the node to compute the partial.	*RPC_*	`(data, *args, **kwargs)`

Warning: Everything that is returned by the`return` statement is sent back to the central vantage6-server. This should never contain any privacy-sensitive information.

Warning: The `client` the master method receives is an `AlgorithmClient` (or a `ContainerClient` if you are using an older version), which is different than the client you use as a user.

For our average algorithm the implementation will look as follows:

```python
import time

from vantage6.tools.util import info

def master(client, data, column_name):
    """Combine partials to global model

    First we collect the parties that participate in the collaboration.
```

(continues on next page)

(continued from previous page)

```
    Then we send a task to all the parties to compute their partial (the
    row count and the column sum). Then we wait for the results to be
    ready. Finally when the results are ready, we combine them to a
    global average.

    Note that the master method also receives the (local) data of the
    node. In most usecases this data argument is not used.

    The client, provided in the first argument, gives an interface to
    the central server. This is needed to create tasks (for the partial
    results) and collect their results later on. Note that this client
    is a different client than the client you use as a user.
    """

    # Info messages can help you when an algorithm crashes. These info
    # messages are stored in a log file which is send to the server when
    # either a task finished or crashes.
    info('Collecting participating organizations')

    # Collect all organization that participate in this collaboration.
    # These organizations will receive the task to compute the partial.
    organizations = client.get_organizations_in_my_collaboration()
    ids = [organization.get("id") for organization in organizations]

    # Request all participating parties to compute their partial. This
    # will create a new task at the central server for them to pick up.
    # We've used a kwarg but is is also possible to use `args`. Although
    # we prefer kwargs as it is clearer.
    info('Requesting partial computation')
    task = client.create_new_task(
        input_={
            'method': 'average_partial',
            'kwargs': {
                'column_name': column_name
            }
        },
        organization_ids=ids
    )

    # Now we need to wait untill all organizations(/nodes) finished
    # their partial. We do this by polling the server for results. It is
    # also possible to subscribe to a websocket channel to get status
    # updates.
    info("Waiting for results")
    task_id = task.get("id")
    task = client.get_task(task_id)
    while not task.get("complete"):
        task = client.get_task(task_id)
        info("Waiting for results")
        time.sleep(1)

    # Once we now the partials are complete, we can collect them.
```

(continues on next page)

(continued from previous page)

```
    info("Obtaining results")
    results = client.get_results(task_id=task.get("id"))

    # Now we can combine the partials to a global average.
    global_sum = 0
    global_count = 0
    for result in results:
        global_sum += result["sum"]
        global_count += result["count"]

    return {"average": global_sum / global_count}

def RPC_average_partial(data, column_name):
    """Compute the average partial

    The data argument contains a pandas-dataframe containing the local
    data from the node.
    """

    # extract the column_name from the dataframe.
    info(f'Extracting column {column_name}')
    numbers = data[column_name]

    # compute the sum, and count number of rows
    info('Computing partials')
    local_sum = numbers.sum()
    local_count = len(numbers)

    # return the values as a dict
    return {
        "sum": local_sum,
        "count": local_count
    }
```

Local testing

Now that we have a vantage6 implementation of the algorithm it is time to test it. Before we run it in a vantage6 setup we can test it locally by using the `ClientMockProtocol` which simulates the communication with the central server.

Before we can locally test it we need to (editable) install the algorithm package so that the Mock client can use it. Simply go to the root directory of your algorithm package (with the `setup.py` file) and run the following:

```
pip install -e .
```

Then create a script to test the algorithm:

```
from vantage6.tools.mock_client import ClientMockProtocol

# Initialize the mock server. The datasets simulate the local datasets from
# the node. In this case we have two parties having two different datasets:
# a.csv and b.csv. The module name needs to be the name of your algorithm
```

(continues on next page)

(continued from previous page)

```
# package. This is the name you specified in `setup.py`, in our case that
# would be v6-average-py.
client = ClientMockProtocol(
    datasets=["local/a.csv", "local/b.csv"],
    module="v6-average-py"
)

# to inspect which organization are in your mock client, you can run the
# following
organizations = client.get_organizations_in_my_collaboration()
org_ids = ids = [organization["id"] for organization in organizations]

# we can either test a RPC method or the master method (which will trigger the
# RPC methods also). Lets start by triggering an RPC method and see if that
# works. Note that we do *not* specify the RPC_ prefix for the method! In this
# example we assume that both a.csv and b.csv contain a numerical column `age`.
average_partial_task = client.create_new_task(
    input_={
        'method':'average_partial',
        'kwargs': {
            'column_name': 'age'
        }
    },
    organization_ids=org_ids
)

# You can directly obtain the result (we dont have to wait for nodes to
# complete the tasks)
results = client.get_results(average_partial_task.get("id"))
print(results)

# To trigger the master method you also need to supply the `master`-flag
# to the input. Also note that we only supply the task to a single organization
# as we only want to execute the central part of the algorithm once. The master
# task takes care of the distribution to the other parties.
average_task = client.create_new_task(
    input_={
        'master': 1,
        'method':'master',
        'kwargs': {
            'column_name': 'age'
        }
    },
    organization_ids=[org_ids[0]]
)
results = client.get_results(average_task.get("id"))
print(results)
```

Building and Distributing

Now that we have a fully tested algorithm for the vantage6 infrastructure. We need to package it so that it can be distributed to the data-stations/nodes. Algorithms are delivered in Docker images. So that's where we need the `Dockerfile` for. To build an image from our algorithm (make sure you have docker installed and it's running) you can run the following command from the root directory of your algorithm project.

```
docker build -t harbor2.vantage6.ai/demo/average .
```

The option `-t` specifies the (unique) identifier used by the researcher to use this algorithm. Usually this includes the registry address (harbor2.vantage6.ai) and the project name (demo).

Note: In case you are using docker hub as registry, you do not have to specify the registry or project as these are set by default to the Docker hub and your docker hub username.

```
docker push harbor2.vantage6.ai/demo/average
```

Note: Reach out to us on Discord if you want to use our registries (harbor.vantage6.ai and harbor2.vantage6.ai).

4.5.2.4 Cross-language serialization

It is possible that a vantage6 algorithm is developed in one programming language, but you would like to run the task from another language. For these use-cases, the Python algorithm wrapper and client support cross-language serialization. By default, input to the algorithms and output back to the client are serialized using pickle. However, it is possible to define a different serialization format.

Input and output serialization can be specified as follows:

```
client.post_task(
    name='mytask',
    image='harbor2.vantage6.ai/testing/v6-test-py',
    collaboration_id=COLLABORATION_ID,
    organization_ids=ORGANIZATION_IDS,
    data_format='json', # Specify input format to the algorithm
    input_={
        'method': 'column_names',
        'kwargs': {'data_format': 'json'}, # Specify output format
    }
)
```

4.6 Technical Docs

4.6.1 Architecture

4.6.1.1 Network Actors

Server

Note: When we refer to the server, this is not just the *vantage6-server*, but also other infrastructure components that the vantage6 server relies on.

The server is responsible for coordinating all communication in the vantage6 network. It consists of several components:

vantage6 server
Contains the users, organizations, collaborations, tasks and their results. It handles authentication and authorization to the system and is the central point of contact for clients and nodes. For more details see *vantage6-server*.

Docker registry
Contains algorithms stored in Images which can be used by clients to request a computation. The node will retrieve the algorithm from this registry and execute it. It is possible to use Docker hub for this, however some (minor) features will not work.

An optional additional feature of the Docker registry would be Docker Notary. This is a service allows verification of the algorithm author.

VPN server (optionally)
Is required if algorithms need to be able to engage in peer-to-peer communication. This is usually the case when working with MPC but can also be useful for other use cases.

RabbitMQ message queue (optionally)
The *vantage6-server* uses the web-sockets protocol to communicate between server, nodes and clients it is impossible to horizontally scale the number of vantage6-server instances. RabbitMQ is used to synchronize the messages between multiple *vantage6-server* instances.

Data Station

vantage6-node
The data station hosts the node (vantage6-node) and a database. The database could be in any format, but not all algorithms support all database types. There is tooling available for CSV, Parquet and SPARQL. There are other data-adapters (e.g. OMOP and FHIR) in development. For more details see *vantage6-node*.

database
The node is responsible for executing the algorithms on the **local data**. It protects the data by allowing only specified algorithms to be executed after verifying their origin. The **vantage6-node** is responsible for picking up the task, executing the algorithm and sending the results back to the server. The node needs access to local data. This data can either be a file (e.g. csv) or a service (e.g. a database).

User or Application

A user or application interacts with the *vantage6-server*. They can create tasks and retrieve their results, or manage entities at the server (i.e. creating or editing users, organizations and collaborations). This can be done using clients or the user-interface. For more details see *vantage6-clients* and *vantage6-UI*.

4.6.1.2 Components

vantage6-server

vantage6-node

vantage6-clients

vantage6-UI

Implementation details are given in the /node/node, /server/server, and /api sections of the documentation.

Note: The following sections are based on our publications:

- VANTAGE6: an open source priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange
- An Improved Infrastructure for Privacy-Preserving Analysis of Patient Data

---

4.6.1.3 Architecture

Vantage6 uses a client-server model, which is shown in `architecture-overview`. In this scenario, the researcher can pose a question and using his/her preferred programming language, send it as a task (also known as computation request) to the (central) server through function calls. The server is in charge of processing the task as well as of handling administrative functions such as authentication and authorization. The requested algorithm is delivered as a container image to the nodes, which have access to their own (local) data. When the algorithm has reached a solution, it is transmitted via the server to the researcher. A more detailed explanation of these components is given as follows.

First, the researcher defines a question. In order to answer it, (s)he identifies which parties possess the required data and establishes a collaboration with them. Then, the parties specify which variables are needed and, more importantly, they agree on their definition. Preferably, this is done following previously established data standards suitable for the field and question at hand. Moreover, it is strongly encouraged that the parties adhere to practices and principles that make their data FAIR (findable, accessible, interoperable, and reusable).

Once this is done, the researcher can pose his/her question as a task to the server in an HTTP request. Vantage6 allows the researcher to do so using any platform of his/her preference (e.g., Python, R, Postman, custom UI, etc.). The request contains a JSON body which includes information about the collaboration and the party for which the request is intended, a reference to a Docker image (corresponding to the selected algorithm), and optional inputs (usually algorithm parameters). By default, the task is sent to all parties.

Vantage6's processing of the task (i.e., server and nodes functionality) occurs behind the scenes. The researcher only needs to deal with his/her working environment (e.g., Jupyter notebook, RStudio).

Once the results are ready, the researcher can obtain them in two ways: on demand (i.e., polling), or through a continuous connection with the server where messages can be sent/received instantly (i.e., WebSocket channel). Due to its speed and efficiency, the latter is preferred.

Fig. 4.8 shows a more detailed diagram of vantage6's server. First, the server is configured by an administrator through a command line interface. The server's parameters (e.g., IP, port, log settings, etc.) are stored into a configuration file. The latter is loaded when the server starts. Once the server is running, entities (e.g., tasks, users, nodes) can be managed through a RESTful API. Furthermore, a WebSocket channel allows communication of simple messages (e.g., status updates) between the different components. This reduces the number of server requests (i.e., neither the researcher nor the nodes need to poll for tasks or results), improving the speed and efficiency of message transmission.

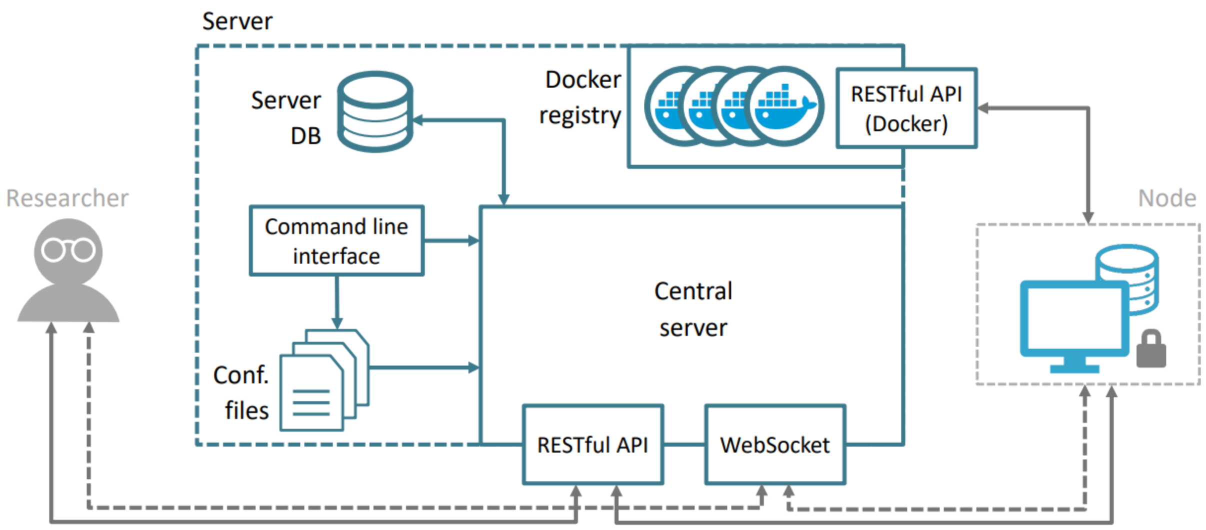


Fig. 4.8: Vantage6's server. An administrator uses the command line interface to configure and start the server. After the server loads its configuration parameters (which are stored in a YAML file), it exposes its RESTful API. It is worth noting that the central server's RESTful API is different from that of the Docker registry.

The central server also stores metadata and information of the researcher (user), parties, collaborations, tasks, nodes, and results. Fig. 4.9 shows its corresponding database model.

A single computation request can lead to many requests to the server, especially when an iterative algorithm is used in combination with many nodes (Assuming the algorithm does not make heavily use of the direct-communication feature). Therefore it is important that the server can handle multiple requests at once. To achief this, the server needs to be able to scale horizontally.

The server and node have a peristent connection through a websocket channel. This complicates the horizontal scalability as nodes can connect to different server instances. E.g. it is not trivial to send a message to all parties when an event occurs in one of the server instances. This problem can be solved by introducing a message broken to which all server instances connect to synchronise all messages.

Algorithm containers can directly communicate (using a ip/port combination) with other algorithm containers in the network using a VPN service. This VPN service needs to be configured in the server as the nodes automatically retrieve the VPN certificates on startup (when the VPN option enabled).

In order for the vantage6-server to retrieve the certificates from the VPN server, this VPN server required to have an API to do so. Therefore the open-source EduVPN solution is used. Which is basically a wrapper arround an OpenVPN instance to provide a feature rich interface.

In order to host a node, the parties need to comply with a few minimal system requirements: Python 3.6+, Docker Community Edition (CE), a stable internet connection, and access to the data. Figure 5 shows a more detailed diagram

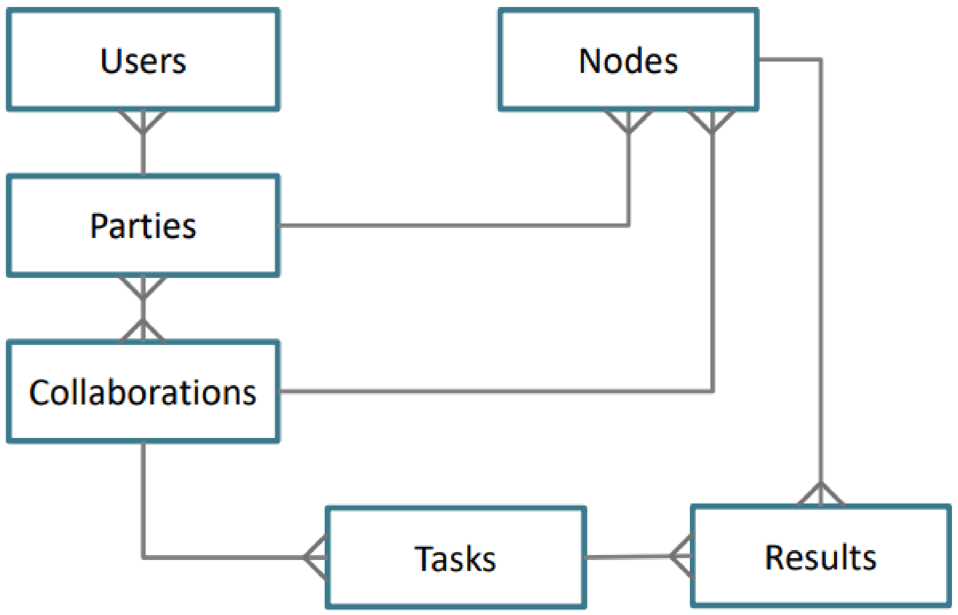


Fig. 4.9: Database model of the central server (Fig. 4.8). The users are always members of a party, which can participate in multiple collaborations. Within a party, users can have different roles (e.g., an administrator is allowed to accept collaborations). For each collaboration a party takes place in, it should create a (running) node. Tasks are always part of a single collaboration and have one or multiple results. In turn, results are always part of a single task and node.

of a single VANTAGE6 node.

In this case, an administrator uses a command line interface to configure the node's core and to start the Docker daemon. We can think of the latter as a service which manages Docker images, containers, volumes, etc. The daemon starts the node's core, which in turn instructs the daemon to create the data volume. The latter contains a copy of the host's data of interest. It is in this moment when the party can exert its autonomy by deciding how much of its data will it allow to contribute to the global solution at hand. After this step, all the pieces are in place for the task execution.

The node receives a task from the server (which could involve a master or an algorithm container) and executes it by downloading the requested (and previously approved) Docker image. The corresponding container accesses the local data through the node and executes the algorithm with the given parameters. Then, the algorithm outputs a set of (intermediate) results, which is sent to the server through the RESTful API. The user or the master container collects these results of all nodes. If needed, it computes a first version of the global solution and sends it back to the nodes, which use it to compute a new set of results. This process could be iteratively until the model's global solution converges or after a fixed number of iterations. This iterative approach is quite generic and allows flexibility by supporting numerous algorithms that deal with horizontally- or vertically-partitioned data.

It is worth emphasizing that the data always stay at their original location It is only intermediate results (i.e., aggregated values, coefficients) that are transmitted, which immensely reduce the risk of leaking private patient information. Furthermore, all messages (node to node, node to user) are end-to-end-encrypted, adding an extra layer of security. It is also worth mentioning that the parties hosting the nodes are allowed to be heterogeneous: as long as they comply with the minimal system requirements, they can have their own hardware and operating system.

4.6.2 Features

The following pages each describe one feature of vantage6 in some detail.

4.6.2.1 Server features

The following pages each describe one feature of the vantage6 server.

Two-factor authentication

Available since version 3.5.0

The vantage6 infrastructure includes the option to use two-factor authentication (2FA). This option is set at the server level: the server administrator decides if it is either enabled or disabled for everyone. Users cannot set this themselves. Server administrators can choose to require 2FA when prompted in `vserver new`, or by adding the option `two_factor_auth: true` to the configuration file (see *Configure*).

Currently, the only 2FA option is to use Time-based one-time passwords (TOTP) With this form of 2FA, you use your phone to scan a QR code using an authenticator app like LastPass authenticator or Google authenticator. When you scan the QR code, your vantage6 account is added to the authenticator app and will show you a 6-digit code that changes every 30 seconds.

Setting up 2FA for a user

If a new user logs in, or if a user logs in for the first time after a server administrator has enabled 2FA, they will be required to set it up. The endpoint `/token/user` will first verify that their password is correct, and then set up 2FA. It does so by generating a random TOTP secret for the user, which is stored in the database. From this secret, a URI is generated that can be used to visualize the QR code.

If the user is logging in via the vantage6 user interface, this QR code will be visualized to allow the user to scan it. Also, users that login via the Python client will be shown a QR code. In both cases, they also have the option to manually enter the TOTP secret into their authenticator app, in case scanning the QR code is not possible.

Users that log in via the R client or directly via the API will have to visualize the QR code themselves, or manually enter the TOTP secret into their authenticator app.

Using 2FA

If a user has already setup 2FA tries to login, the endpoint `/token/user` will require that they provide their 6-digit TOTP code via the `mfa_code` argument. This code will be checked using the TOTP secret stored in the database, and if it is valid, the user will be logged in.

To prevent users with a slow connection from having difficulty logging in, valid codes from the 30s period directly prior to the current period will also be logged in.

Resetting 2FA

When a user loses access to their 2FA, they may reset it via their email. They should use the endpoint `/recover/2fa/lost` to get an email with a reset token and then use the reset token in `/recover/2fa/reset` to reset 2FA. This endpoint will give them a new QR code that they can visualize just like the initial QR code.

SSH Tunnel

Available since version 3.7.0

Vantage6 algorithms are normally disconnected from the internet, and are therefore unable to connect to access data that is not connected to the node on node startup. Via this feature, however, it is possible to connect to a remote server through a secure SSH connection. This allows you to connect to a dataset that is hosted on another machine than your node, as long as you have SSH access to that machine.

Setting up SSH tunneling

1. Create a new SSH key pair

Create a new key pair *without* a password on your node machine. To do this, enter the command below in your terminal, and leave the password empty when prompted.

```
ssh-keygen -t rsa
```

You are required not to use a password for the private key, as vantage6 will set up the SSH tunnel without user intervention and you will therefore not be able to enter the password in that process.

2. Add the public key to the remote server

Copy the contents of the public key file (`your_key.pub`) to the remote server, so that your node will be allowed to connect to it. In the most common case, this means adding your public key to the `~/.ssh/authorized_keys` file on the remote server.

3. Add the SSH tunnel to your node configuration

An example of the SSH tunnel configuration can be found below. See *here* for a full example of a node configuration file.

```
databases:
  httpserver: http://my_http:8888
ssh-tunnels:
  - hostname: my_http
    ssh:
      host: my-remote-machine.net
      port: 22
      fingerprint: "ssh-rsa AAAAE2V....wwef987vD0="
      identity:
        username: bob
        key: /path/to/your/private/key
```

(continues on next page)

(continued from previous page)

```
    tunnel:
      bind:
        ip: 0.0.0.0
        port: 8888
      dest:
        ip: 127.0.0.1
        port: 9999
```

There are a few things to note about the SSH tunnel configuration:

1. You can provide multiple SSH tunnels in the `ssh-tunnels` list, by simply specifying more blocks of the same format.
2. The hostname of each tunnel should come back in one of the databases, so that they may be accessible to the algorithms.
3. The `host` is the address at which the remote server can be reached. This is usually an IP address or a domain name. Note that you are able to specify IP addresses in the local network. Specifying non-local IP addresses is not recommended, as you might be exposing your node if the IP address is spoofed.
4. The `fingerprint` is the fingerprint of the remote server. You can usually find it in */etc/ssh/ssh_host_rsa_key.pub* on the remote server.
5. The `identity` section contains the username and path to the private key your node is using. The username is the username you use to log in to the remote server, in the case above it would be `ssh bob@my-remote-machine.net`.
6. The `tunnel` section specifies the port on which the SSH tunnel will be listening, and the port on which the remote server is listening. In the example above, on the remote machine, there would be a service listening on port 9999 on the machine itself (which is why the IP is 127.0.0.1 a.k.a. localhost). The tunnel will be bound to port 8888 on the node machine, and you should therefore take care to include the correct port in your database path.

Using the SSH tunnel

How you should use the SSH tunnel depends on the service that you are running on the other side. In the example above, we are running a HTTP server and therefore we should obtain data via HTTP requests. In the case of a SQL service, one would need to send SQL queries to the remote server instead.

Note: We aim to extend this section later with an example of an algorithm that is using this feature.

Horizontal scaling

By horizontal scaling, we mean that you can run multiple instances of the vantage6 server simultaneously to handle a high workload. This is useful when a single machine running the server is no longer sufficient to handle all requests.

How it works

Horizontal scaling with vantage6 can be done using a RabbitMQ server. RabbitMQ is a widely used message broker. Below, we will first explain how we use RabbitMQ, and then discuss the implementation.

The websocket connection between server and nodes is used to process various changes in the network's state. For example, a node can create a new (sub)task for the other nodes in the collaboration. The server then communicates these tasks via the socket connection. Now, if we use multiple instances of the central server, different nodes in the same collaboration may connect to different instances, and then, the server would not be able to deliver the new task properly. This is where RabbitMQ comes in.

When RabbitMQ is enabled, the websocket messages are directed over the RabbitMQ message queue, and delivered to the nodes regardless of which server instance they are connected to. The RabbitMQ service thus helps to ensure that all websocket events are still communicated properly to all involved parties.

How to use

If you use multiple server instances, you should always connect them to the same RabbitMQ instance. You can achieve this by adding your RabbitMQ server when you create a new server with `vserver new`, or you can add it later to your server configuration file as follows:

```
rabbitmq_uri: amqp://$user:$password@$host:$port/$vhost
```

Where `$user` is the username, `$password` is the password, `$host` is the URL where your RabbitMQ service is running, `$port` is the queue's port (which is 5672 if you are using the RabbitMQ Docker image), and `$vhost` is the name of your virtual host (you could e.g. run one instance group per vhost).

Deploy

If you are running a test server with `vserver start`, a RabbitMQ docker container will be started automatically for you. This docker container contains a management interface which will be available on port 15672.

For deploying a production server, there are several options to run RabbitMQ. For instance, you can install *RabbitMQ on Azure <https://www.golinuxcloud.com/install-rabbitmq-on-azure/>_*.

Permission management

Almost every endpoint on the API is under role-based access control: not everyone and everything is allowed to access it.

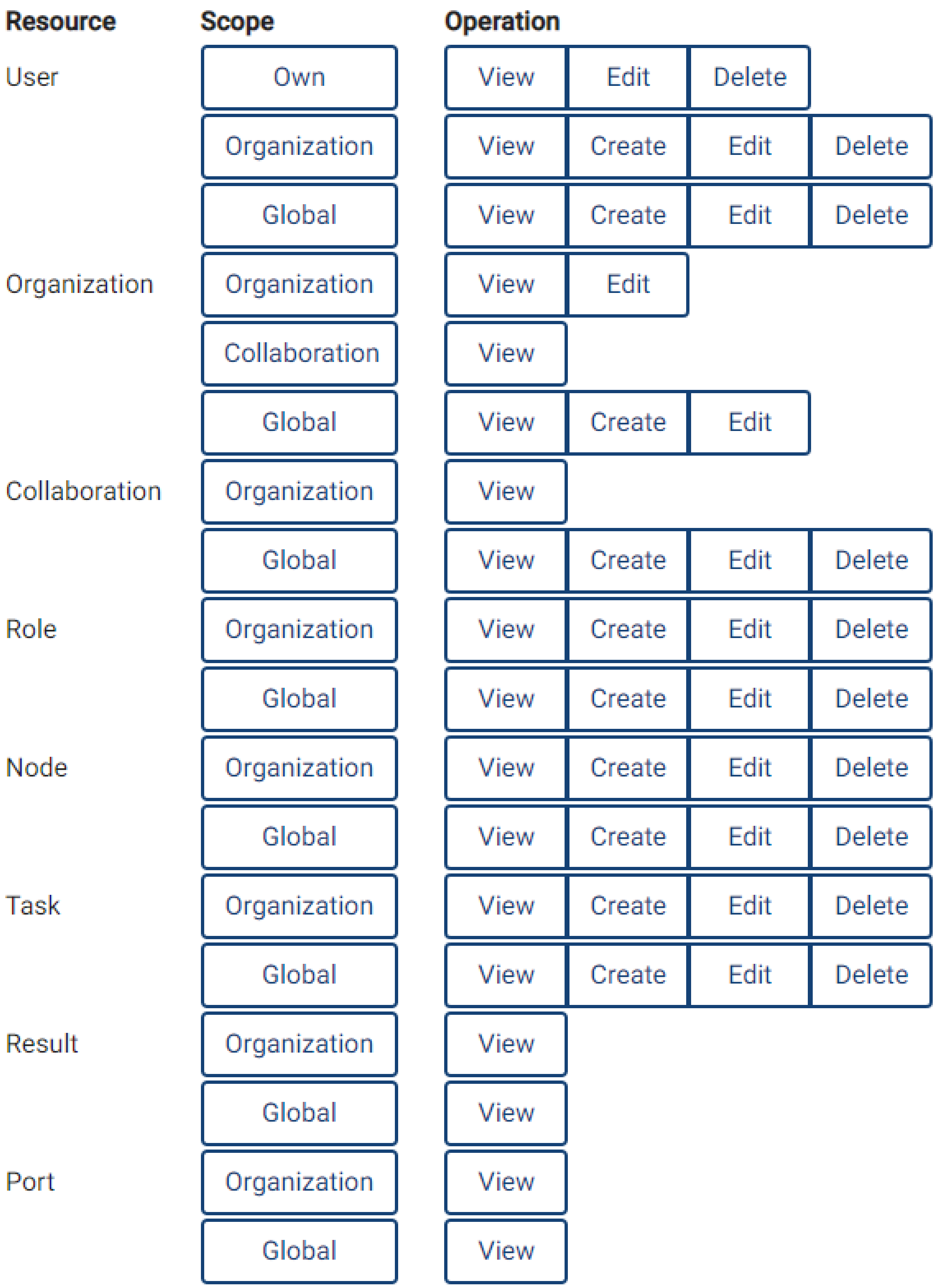
There are three types of entities that can attempt to access the API: users, nodes and algorithm containers. Not every endpoint is available to all three entities. Therefore, there are decorators such as:

- `@only_for(['user', 'container']`: only accessible for users and algorithm containers
- `@with_user`: only users have access to this endpoint

These decorators ensure that only authenticated entities of the right type can enter the endpoint.

When an endpoint is then entered, there are additional permission checks. For users, permissions vary per user. Nodes and algorithm containers all have the same permissions, but for specific situations there are specific checks. For instance, nodes are only allowed to update their own results, and not those of other nodes. These checks are performed within the endpoints themselves.

The following rules are defined:



Resource	Scope	Operation			
User	Own	View	Edit	Delete	
	Organization	View	Create	Edit	Delete
	Global	View	Create	Edit	Delete
Organization	Organization	View	Edit		
	Collaboration	View			
	Global	View	Create	Edit	
Collaboration	Organization	View			
	Global	View	Create	Edit	Delete
Role	Organization	View	Create	Edit	Delete
	Global	View	Create	Edit	Delete
Node	Organization	View	Create	Edit	Delete
	Global	View	Create	Edit	Delete
Task	Organization	View	Create	Edit	Delete
	Global	View	Create	Edit	Delete
Result	Organization	View			
	Global	View			
Port	Organization	View			
	Global	View			

Fig. 4.10: The rules that are available per resource, scope, and operation. For example, the first rule with resource ‘User’, scope ‘Own’ and operation ‘View’ will allow a user to view their own user details.

The rules have an operation, a scope, and a resource that they work on. For instance, a rule with operation 'View', scope 'Organization' and resource 'Task', will allow a user to view all tasks of their own organization. There are 4 operations (view, edit, create and delete) that correspond to GET, PATCH, CREATE and DELETE requests, respectively. The scopes are:

- Global: all resources of all organizations
- Organization: resources of the user's own organization
- Collaboration: resources of all organizations that the user's organization is in a collaboration with
- Own: these are specific to the user endpoint. Permits a user to see/edit their own user, but not others within the organization.

API response structure

Each API endpoint returns a JSON response. All responses are structured in the same way, according to the HATEOAS constraints. An example is detailed below:

```
>>> client.task.get(task_id)
{
    "id": 1,
    "name": "test",
    "results": [
        {
            "id": 2,
            "link": "/api/result/2",
            "methods": [
                "PATCH",
                "GET"
            ]
        }
    ],
    "image": "harbor2.vantage6.ai/testing/v6-test-py",
    ...
}
```

The response for this task includes the results that are attached to this task. In compliance with HATEOAS, a link is supplied to the link where the result can be viewed in more detail.

4.6.2.2 Node features

The following pages each describe one feature of the vantage6 node.

** Under construction ** ..

4.6.2.3 Algorithm features

The following pages each describe one feature of vantage6 algorithms.

Algorithm wrappers

Algorithm wrappers are used in algorithms to make it easier for algorithms to handle input and output.

- list the available wrappers
- links to their docstrings

4.6.2.4 Communication between components

The following pages each describe one way that is used to communicate between different vantage6 components.

SocketIO connection

A SocketIO connection is a bidirectional, persistent, event-based communication line. In vantage6, it is used for example to send status updates from the server to the nodes or to send a signal to a node that it should kill a task.

Each socketIO connection consists of a server and one or more clients. The clients can only send a message to the server and not to each other. The server can send messages to all clients or to a specific client. In vantage6, the central server is the socketIO server; the clients can be nodes or users.

Note: The vantage6 user interface automatically establishes a socketIO connection with the server when the user logs in. The user can then view the updates they are allowed to see.

Permissions

The socketIO connection is split into different rooms. The vantage6 server decides which rooms a client is allowed to join; they will only be able to read messages from that room.

Nodes always join the room of their own collaboration, and a room of all nodes. Users only join the room of collaborations whose events they are allowed to view which is checked via event view rules.

Usage in vantage6

The server sends the following events to the clients:

- Notify nodes a new task is available
- Letting nodes and users know if a node in their collaboration comes online or goes offline
- Instructing nodes to renew their token if it is expired
- Letting nodes and users know if a task changed state on a node (e.g. started, finished, failed). This is especially important for nodes to know in case an algorithm they are running depends on the output of another node.
- Instruct nodes to kill one or more tasks
- Checking if nodes are still alive

The nodes send the following events to the server:

- Alert the server of task state changes (e.g. started, finished, failed)
- Share information about the node configuration (e.g. which algorithms are allowed to run on the node)

In theory, users could use their socketIO connection to send events, but none of the events they send will lead to action on the server.

Algorithm-to-algorithm comunication

Since version 3.0.0

Originally, all communication in the vantage6 infrastructure occurs via the central server. Algorithms and nodes could not directly communicate with one another. Since version 3.0.0, algorithms can communicate with one another directly, without the need to go through the central server. This is achieved by connecting the nodes to a VPN network.

The implementation of algorithm-to-algorithm communication in vantage6 is discussed at length in this paper.

When to use

Some algorithms require a lot of communication between algorithm containers before a solution is achieved. For example, there are algorithms that uses iterative methods to optimize a solution, or algorithms that share partial machine learning models with one another in the learning process.

For such algorithms, using the default communication method (via the central server) can be very inefficient. Also, some popular libraries assume that direct communication between algorithm containers is possible. These libraries would have to be adapted specifically for the vantage6 infrastructure, which is not always feasible. In such cases, it is better to setup a VPN connection to allow algorithm containers to communicate directly with one another.

Another reason to use a VPN connection is that for some algorithms, routing all partial results through the central server can be undeseriable. For example, with many algorithms using an MPC protocol, it may be possible for the central party to reconstruct the original data if they have access to all partial results.

How to use

In order to use a VPN connection, a VPN server must be set up, and the vantage6 server and nodes must be configured to use this VPN server. Below we detail How this can be done.

Installing a VPN server

To use algorithm-to-algorithm communication, a VPN server must be set up by the server administrator. The installation instructions for the VPN server are *here*.

Configuring the vantage6 server

The vantage6 server must be configured to use the VPN server. This is done by adding the following configuration snippet to the configuration file.

```yaml
vpn_server:
    # the URL of your VPN server
    url: https://your-vpn-server.ext

    # OATH2 settings, make sure these are the same as in the
    # configuration file of your EduVPN instance
    redirect_url: http://localhost
    client_id: your_VPN_client_user_name
    client_secret: your_VPN_client_user_password

    # Username and password to acccess the EduVPN portal
    portal_username: your_eduvpn_portal_user_name
    portal_userpass: your_eduvpn_portal_user_password
```

Note that the vantage6 server does not connect to the VPN server itself. It uses the configuration above to provide nodes with a VPN configuration file when they want to connect to VPN.

Configuring the vantage6 node

A node administrator has to configure the node to use the VPN server. This is done by adding the following configuration snippet to the configuration file.

```yaml
vpn_subnet: '10.76.0.0/16'
```

This snippet should include the subnet on which the node will connect to the VPN network, and should be part of the subnet range of the VPN server. Node administrators should consult the VPN server administrator to determine which subnet range to use.

If all configuration is properly set up, the node will automatically connect to the VPN network on startup.

Warning: If the node fails to connect to the VPN network, the node will not stop. It will print a warning message and continue to run.

Note: Nodes that connect to a vantage6 server with VPN do not necessarily have to connect to the VPN server themselves: they may be involved in a collaboration that does not require VPN.

How to test the VPN connection

This algorithm can be used to test the VPN connection. The script *test_on_v6.py* in this repository can be used to send a test task which will print whether echoes over the VPN network are working.

Use VPN in your algorithm

If you are using the Python algorithm client, you can call the following function:

```
client.vpn.get_addresses()
```

which will return a dictionary containing the VPN IP address and port of each of the algorithms running that task.

Warning: If you are using the old algorithm client `ContainerClient` (which is the default in vantage6 3.x), you should use `client.get_algorithm_addresses()` instead.

If you are not using the algorithm client, you can send a request to to the endpoint `/vpn/algorithm/addresses` on the vantage6 server (via the node proxy server), which will return a dictionary containing the VPN IP address and port of each of the algorithms running that task.

How does it work?

As mentioned before, the implementation of algorithm-to-algorithm communication is discussed at length in this paper. Below, we will give a brief overview of the implementation.

On startup, the node requests a VPN configuration file from the vantage6 server. The node first checks if it already has a VPN configuration file and if so, it will try to use that. If connecting with the existing configuration file fails, it will try to renew the configuration file's keypair by calling `/vpn/update`. If that fails, or if no configuration file is present yet (e.g. on first startup of a node), the node will request a new configuration file by calling `/vpn`.

The VPN configuration file is an `.ovpn` file that is passed to a VPN client container that establishes the VPN connection. This VPN client container keeps running in the background for as long as the node is running.

When the VPN client container is started, a few network rules are changed on the host machine to forward the incoming traffic on the VPN subnet to the VPN client container. This is necessary because the VPN traffic will otherwise never reach the vantage6 containers. The VPN client container is configured to drop any traffic that does not originate from the VPN connection.

When a task is started, the vantage6 node determines how many ports that particular algorithm requires on the local Docker network. It determines which ports are available and then assigns those ports to the algorithm. The node then stores the VPN IP address and the assigned ports in the database. Also, it configures the local Docker network such that the VPN client container forwards all incoming traffic for algorithm containers to the right port on the right algorithm container. *Vice versa*, the VPN client container is configured to forward outgoing traffic over the VPN network to the right addresses.

Only when the all this configuration is completed, is the algorithm container started.

4.6.3 Node

Below you will find the structure of the classes and functions that comprise the node. A few that we would like to highlight:

- *Node*: the main class in a vantage6 node.
- *NodeContext* and *DockerNodeContext*: classes that handle the node configuration. The latter inherits from the former and adds some properties for when the node runs in a docker container.
- *DockerManager*: Manages the docker containers and networks of the vantage6 node.
- *DockerTaskManager*: Start a docker container that runs an algorithm and manage its lifecycle.
- *VPNManager*: Sets up the VPN connection (if it is configured) and manages it.
- *vnode-local commands*: commands to run non-dockerized (development) instances of your nodes.

4.6.3.1 `Node` class

4.6.3.2 `NodeContext` class

`class NodeContext`(*instance_name*, *environment='application'*, *system_folders=False*, *config_file=None*)

Node context

See DockerNodeContext for the node instance mounts when running as a dockerized service.

Parameters

- **`instance_name`** (`str`) – Name of the configuration instance, corresponds to the filename of the configuration file.
- **`environment`** (`str, optional`) – DTAP environment to be loaded, by default N_ENV
- **`system_folders`** (`bool, optional`) – _description_, by default N_FOL
- **`config_file`** (`str, optional`) – _description_, by default None

`INST_CONFIG_MANAGER`

alias of `NodeConfigurationManager`

`classmethod available_configurations`(*system_folders=False*)

Find all available server configurations in the default folders.

Parameters

`system_folders` (`bool, optional`) – System wide or user configuration, by default N_FOL

Returns

The first list contains validated configuration files, the second list contains invalid configuration files.

Return type

Tuple[List, List]

`classmethod config_exists`(*instance_name*, *environment='application'*, *system_folders=False*)

Check if a configuration file exists.

Parameters

- `instance_name` (*str*) – Name of the configuration instance, corresponds to the filename of the configuration file.
- `environment` (*str, optional*) – DTAP environment that needs to be present, by default N_ENV
- `system_folders` (*bool, optional*) – System wide or user configuration, by default N_FOL

Returns
Whether the configuration file exists or not

Return type
bool

`property databases`

Dictionary of local databases that are available for this node.

Returns
dictionary with database names as keys and their corresponding paths as values.

Return type
dict

`property docker_container_name: str`

Unique Docker container name of the node.

Returns
Unique Docker container name

Return type
str

`property docker_network_name: str`

Private Docker network name which is unique for this node.

Returns
Docker network name

Return type
str

`property docker_ssh_volume_name: str`

Docker volume in which the SSH configuration is stored.

Returns
Docker voluem name

Return type
str

`docker_temporary_volume_name`(*run_id*)

Docker volume in which temporary data is stored. Temporary data is linked to a specific run. Multiple algorithm containers can have the same run id, and therefore the share same temporary volume.

Parameters
`run_id` (*int*) – run id provided by the server

Returns
Docker volume name

Return type
str

`property docker_volume_name: str`

Docker volume in which task data is stored. In case a file based database is used, this volume contains the database file as well.

Returns
Docker volume name

Return type
str

`property docker_vpn_volume_name: str`

Docker volume in which the VPN configuration is stored.

Returns
Docker volume name

Return type
str

`classmethod from_external_config_file`(*path*, *environment='application'*, *system_folders=False*)

Create a node context from an external configuration file. External means that the configuration file is not located in the default folders but its location is specified by the user.

Parameters

- `path` (*str*) – Path of the configuration file
- `environment` (*str, optional*) – DTAP environment to be loaded, by default N_ENV
- `system_folders` (*bool, optional*) – System wide or user configuration, by default N_FOL

Returns
Node context object

Return type
NodeContext

`get_database_uri`(*label='default'*)

Obtain the database URI for a specific database.

Parameters
`label` (*str, optional*) – Database label, by default “default”

Returns
URI to the database

Return type
str

`static type_data_folder`(*system_folders=False*)

Obtain OS specific data folder where to store node specific data.

Parameters
`system_folders` (*bool, optional*) – System wide or user configuration, by default N_FOL

Returns
Path to the data folder

Return type
Path

4.6.3.3 `DockerNodeContext` class

4.6.3.4 `DockerBaseManager` class

4.6.3.5 `DockerManager` class

4.6.3.6 `DockerTaskManager` class

4.6.3.7 `VPNManager` class

4.6.3.8 Algorithm execution exceptions

4.6.3.9 Proxy server

4.6.3.10 `vnode-local` commands

4.6.4 Server

The server has a central function in the vantage6 architecture. It stores in the database which organizations, collaborations, users, etc. exist. It allows the users and nodes to authenticate and subsequently interact through the API the server hosts. Finally, it also communicates with authenticated nodes and users via the socketIO server that is run here.

4.6.4.1 Main server class

class ServerApp(*ctx*)

Vantage6 server instance.

Variables
ctx (*ServerContext*) – Context object that contains the configuration of the server.

configure_api()

Define global API output and its structure.

configure_flask()

Configure the Flask settings of the vantage6 server.

configure_jwt()

Configure JWT authentication.

static configure_logging()

Set third party loggers to a warning level

load_resources()

Import the modules containing API resources.

start()

Start the server.

Before server is really started, some database settings are checked and (re)set where appropriate.

4.6.4.2 Starting the server

`run_server`(*config*, *environment='prod'*, *system_folders=True*)

Run a vantage6 server.

Parameters

- **config** (*str*) – Configuration file path
- **environment** (*str*) – Configuration environment to use.
- **system_folders** (*bool*) – Whether to use system or user folders. Default is True.

Returns

A running instance of the vantage6 server

Return type

ServerApp

Warning: Note that the `run_server` function is normally not used directly to start the server, but is used as utility function in places that start the server. The recommended way to start a server is using uWSGI as is done in `vserver start`.

`run_dev_server`(*server_app*, **args*, ***kwargs*)

Run a vantage6 development server (outside of a Docker container).

Parameters

server_app (`ServerApp`) – Instance of a vantage6 server

Return type

`None`

4.6.4.3 Permission management

`class Scope`(*value*)

Enumerator of all available scopes

`COLLABORATION = 'col'`

`GLOBAL = 'glo'`

`ORGANIZATION = 'org'`

`OWN = 'own'`

`class Operation`(*value*)

Enumerator of all available operations

`CREATE = 'c'`

`DELETE = 'd'`

`EDIT = 'e'`

`VIEW = 'v'`

class RuleCollection(*name*)

Class that tracks a set of all rules for a certain resource name

Parameters
name (*str*) – Name of the resource endpoint (e.g. node, organization, user)

add(*scope*, *operation*)

Add a rule to the rule collection

Parameters

- **scope** (Scope) – Scope within which to apply the rule
- **operation** (Operation) – What operation the rule applies to

Return type
None

class PermissionManager

Loads the permissions and syncs rules in database with rules defined in the code

appender(*name*)

Add a module's rules to the rule collection

Parameters
name (*str*) – The name of the module whose rules are to be registered

Returns
A callable `register_rule` function

Return type
Callable

assign_rule_to_container(*resource*, *scope*, *operation*)

Assign a rule to the container role.

Parameters

- **resource** (*str*) – Resource that the rule applies to
- **scope** (Scope) – Scope that the rule applies to
- **operation** (Operation) – Operation that the rule applies to

Return type
None

static assign_rule_to_fixed_role(*fixedrole*, *resource*, *scope*, *operation*)

Attach a rule to a fixed role (not adjustable by users).

Parameters

- **fixedrole** (*str*) – Name of the fixed role that the rule should be added to
- **resource** (*str*) – Resource that the rule applies to
- **scope** (Scope) – Scope that the rule applies to
- **operation** (Operation) – Operation that the rule applies to

Return type
None

`assign_rule_to_node`(*resource*, *scope*, *operation*)

Assign a rule to the Node role.

Parameters

- **`resource`** (*`str`*) – Resource that the rule applies to
- **`scope`** (`Scope`) – Scope that the rule applies to
- **`operation`** (`Operation`) – Operation that the rule applies to

Return type

`None`

`assign_rule_to_root`(*name*, *scope*, *operation*)

Assign a rule to the root role.

Return type

`None`

resource: str

Resource that the rule applies to

scope: Scope

Scope that the rule applies to

operation: Operation

Operation that the rule applies to

`collection`(*name*)

Get a RuleCollection object. If it doesn't exist yet, it will be created.

Parameters

`name` (*`str`*) – Name of the module whose RuleCollection is to be obtained or created

Returns

The collection of rules belonging to the module name

Return type

RuleCollection

`load_rules_from_resources`()

Collect all permission rules from all registered API resources

Return type

`None`

`register_rule`(*resource*, *scope*, *operation*, *description=None*, *assign_to_node=False*, *assign_to_container=False*)

Register a permission rule in the database.

If a rule already exists, nothing is done. This rule can be used in API endpoints to determine if a user, node or container can do a certain operation in a certain scope.

Parameters

- **`resource`** (*`str`*) – API resource that the rule applies to
- **`scope`** (`Scope`) – Scope of the rule
- **`operation`** (`Operation`) – Operation of the rule

- **description** (*String, optional*) – Human readable description where the rule is used for, by default None
- **assign_to_node** (*bool, optional*) – Whether rule should be assigned to the node role or not. Default False
- **assign_to_container** (*bool, optional*) – Whether rule should be assigned to the container role or not. Default False

Return type
None

`static rule_exists_in_db`(*name*, *scope*, *operation*)

Check if the rule exists in the DB.

Parameters

- **name** (*str*) – Name of the rule
- **scope** (Scope) – Scope of the rule
- **operation** (Operation) – Operation of the rule

Returns
Whenever this rule exists in the database or not

Return type
bool

`static verify_user_rules`(*rules*)

Check if an user, node or container has all the *rules*

Parameters
rules (*List[Rule]*) – List of rules that user is checked to have

Returns
False if user has all rules, else a dict with a message

Return type
Union[dict, bool]

4.6.4.4 Socket functionality

`class DefaultSocketNamespace`(*namespace=None*)

This is the default SocketIO namespace. It is used for all the long-running socket communication between the server and the clients. The clients of the socket connection are nodes and users.

When socket communication is received from one of the clients, the functions in this class are called to execute the corresponding action.

`on_algorithm_status_change`(*data*)

An algorithm container has changed its status. This status change may be that the algorithm has finished, crashed, etc. Here we notify the collaboration of the change.

Parameters
data (*Dict*) – Dictionary containing parameters on the updated algorithm status. It should look as follows:

{
# node_id where algorithm container was running "node_id": 1, # new status of algorithm container "status": "active", # result_id for which the algorithm was running "result_id": 1, # collaboration_id for which the algorithm was running "collaboration_id": 1

}

Return type
`None`

`on_connect()`

A new incoming connection request from a client.

New connections are authenticated using their JWT authorization token which is obtained from the REST API. A session is created for each connected client, and lives as long as the connection is active. Each client is assigned to rooms based on their permissions.

Nodes that are connecting are also set to status 'online'.

Return type
`None`

Note: Note that reconnecting clients are treated the same as new clients.

`on_disconnect()`

Client that disconnects is removed from all rooms they were in.

If nodes disconnect, their status is also set to offline and users may be alerted to that. Also, any information on the node (e.g. configuration) is removed from the database.

Return type
`None`

`on_error(`*e*`)`

An receiving an error from a client, log it.

Parameters
`e` (*`str`*) – Error message that is being displayed in the server log

Return type
`None`

`on_message(`*message*`)`

On receiving a message from a client, log it.

Parameters
`message` (*`str`*) – Message that is going to be displayed in the server log

Return type
`None`

`on_node_info_update(`*node_config*`)`

A node sends information about its configuration and other properties. Store this in the database for the duration of the node's session.

Parameters
`node_config` (*`dict`*) – Dictionary containing the node's configuration.

Return type
`None`

`on_ping()`

A client sends a ping to the server. The server detects who sent the ping and sets them as online.

Return type
`None`

4.6.4.5 API endpoints

Warning: The API endpoints are also documented on the `/apidocs` endpoint of the server (e.g. `https://petronas.vantage6.ai/apidocs`). We are therefore not including the API documentation here. Instead, we merely list the supporting functions and classes.

4.6.4.6 SQLAlchemy models

Helper (base) classes

class Database(**args*, ***kwargs*)

Database class that is used to connect to the database and create the database session.

The database is created as a singleton, so that it can be destroyed (as opposed to a module). This is especially useful when creating unit tests in which we want fresh databases every now and then.

add_col_to_table(*column*, *table_cls*)

Database operation to add column to *Table*

Parameters

- **column** (`Column`) – The SQLAlchemy model column that is to be added
- **table_cls** (`Table`) – The SQLAlchemy table to which the column is to be added

Return type

`None`

add_missing_columns()

Check database tables to see if columns are missing that are described in the SQLAlchemy models, and add the missing columns

Return type

`None`

clear_data()

Clear all data from the database.

close()

Delete all tables and close the database connection. Only used for unit testing.

connect(*uri='sqlite:////tmp/test.db'*, *allow_drop_all=False*)

Connect to the database.

Parameters

- **uri** (`str`) – URI of the database. Defaults to a sqlite database in /tmp.
- **allow_drop_all** (`bool, optional`) – If True, the database can be dropped. Defaults to False.

drop_all()

Drop all tables in the database.

get_non_existing_columns(*table_cls*, *table_name*)

Return a list of columns that are defined in the SQLAlchemy model, but are not present in the database

Parameters

- **table_cls** (*Table*) – The table that is evaluated
- **table_name** (*str*) – The name of the table

Returns
List of SQLAlchemy Column objects that are present in the model, but not in the database

Return type
List[Column]

`static is_column_missing(`*column*, *column_names*, *table_name*`)`

Check if column is missing in the table

Parameters

- **column** (*Column*) – The column that is evaluated
- **column_names** (*List[str]*) – A list of all column names in the table
- **table_name** (*str*) – The name of the table the column resides in

Returns
True if column is not in the table or a parent table

Return type
boolean

`class DatabaseSessionManager`

Class to manage DB sessions.

There are 2 different ways a session can be obtained. Either a session used within a request or a session used elsewhere (e.g. socketIO event, iPython or within the application itself).

In case of the Flask request, the session is stored in the flask global *g*. Then, it can be accessed in every endpoint.

In all other cases the session is attached to the db module.

`static clear_session()`

Clear the session. If we are in a flask request, the session is cleared from the flask global *g*. Otherwise, the session is removed from the db module.

Return type
`None`

`static get_session()`

Get a session. Creates a new session if none exists.

Returns
A database session

Return type
Session

`static in_flask_request()`

Check if we are in a flask request.

Returns
True if we are in a flask request, False otherwise

Return type
boolean

`static new_session()`

Create a new session. If we are in a flask request, the session is stored in the flask global *g*. Otherwise, the session is stored in the db module.

Return type
`None`

`class ModelBase`

Declarative base that defines default attributes. All data models inherit from this class.

`delete()`

Delete the object from the database.

Return type
`None`

`classmethod get(`*id_=None*`)`

Get a single object by its id, or a list of objects when no id is specified.

Parameters
id (*int, optional*) – The id of the object to get. If not specified, return all.

`classmethod help()`

Print a help message for the class.

Return type
`None`

`save()`

Save the object to the database.

Return type
`None`

Database models for the API resources

`class AlgorithmPort(`***kwargs*`)`

Table that describes which algorithms are reachable via which ports

Each algorithm with a VPN connection can claim multiple ports via the Dockerfile `EXPOSE` and `LABEL` commands. These claims are saved in this table. Each algorithm container belongs to a single `Result`.

Variables

- **port** (*int*) – The port number that is claimed by the algorithm
- **result_id** (*int*) – The id of the `Result` that this port belongs to
- **label** (*str*) – The label that is claimed by the algorithm
- **result** (*Result*) – The `Result` that this port belongs to

`class Authenticatable(`***kwargs*`)`

Parent table of database entities that can authenticate.

Entities that can authenticate are nodes and users. Containers can also authenticate but these are authenticated indirectly through the nodes.

`static hash(`*secret*`)`

Hash a secret using bcrypt.

Parameters
`secret` (*str*) – Secret to be hashed

Returns
Hashed secret

Return type
str

`class Collaboration(`***kwargs*`)`

Table that describes which collaborations are available.

Collaborations are combinations of one or more organizations that do studies together. Each *Organization* has a *Node* for each collaboration that it is part of. Within a collaboration multiple *Task* can be executed.

Variables

- `name` (*str*) – Name of the collaboration
- `encrypted` (*bool*) – Whether the collaboration is encrypted or not
- `organizations` (list[*Organization*]) – List of organizations that are part of this collaboration
- `nodes` (list[*Node*]) – List of nodes that are part of this collaboration
- `tasks` (list[*Task*]) – List of tasks that are part of this collaboration

`classmethod find_by_name(`*name*`)`

Find *Collaboration* by its name.

Note: If multiple collaborations share the same name, the first collaboration found is returned.

Parameters
`name` (*str*) – Name of the collaboration

Returns
Collaboration with the given name, or None if no collaboration with the given name exists.

Return type
Union[*Collaboration*, None]

`get_node_from_organization(`*organization*`)`

Returns the node that is part of the given *Organization*.

Parameters
`organization` (Organization) – Organization

Returns
Node for the given organization for this collaboration, or None if there is no node for the given organization.

Return type
Union[*Node*, None]

get_nodes_from_organizations(*ids*)

Returns a subset of nodes that are part of the given organizations.

Parameters
ids (*list[int]*) – List of organization ids

Returns
List of nodes that are part of the given organizations

Return type
list[*Node*]

get_organization_ids()

Returns a list of organization ids that are part of this collaboration.

Returns
List of organization ids

Return type
list[int]

get_task_ids()

Returns a list of task ids that are part of this collaboration.

Returns
List of task ids

Return type
list[int]

classmethod name_exists(*name*)

Check if a collaboration with the given name exists.

Parameters
name (*str*) – Name of the collaboration

Returns
True if a collaboration with the given name exists, else False

Return type
bool

class Node(***kwargs*)

Bases: *Authenticatable*

Table that contains all registered nodes.

Variables

- **id** (*int*) – Primary key
- **name** (*str*) – Name of the node
- **api_key** (*str*) – API key of the node
- **collaboration** (*Collaboration*) – Collaboration that the node belongs to
- **organization** (*Organization*) – Organization that the node belongs to

check_key(*key*)

Checks if the provided key matches the stored key.

Parameters
key (*str*) – The key to check

Returns
True if the provided key matches the stored key, False otherwise

Return type
bool

`classmethod exists(`*organization_id*, *collaboration_id*`)`

Check if a node exists for the given organization and collaboration.

Parameters

- `organization_id` (*int*) – The id of the organization
- `collaboration_id` (*int*) – The id of the collaboration

Returns
True if a node exists for the given organization and collaboration, False otherwise.

Return type
bool

`classmethod get_by_api_key(`*api_key*`)`

Returns Node based on the provided API key.

Parameters
`api_key` (*str*) – The API key of the node to search for

Returns
Returns the node if a node is associated with api_key, None if no node is associated with api_key.

Return type
Node | None

`classmethod get_online_nodes()`

Return nodes that currently have status 'online'

Returns
List of node models that are currently online

Return type
list[*Node*]

`class Organization(`***kwargs*`)`

Table that describes which organizations are available.

An organization is the legal entity that plays a central role in managing distributed tasks. Each organization contains a public key which other organizations can use to send encrypted messages that only this organization can read.

Variables

- `name` (*str*) – Name of the organization
- `domain` (*str*) – Domain of the organization
- `address1` (*str*) – Address of the organization
- `address2` (*str*) – Address of the organization
- `zipcode` (*str*) – Zipcode of the organization
- `country` (*str*) – Country of the organization
- `_public_key` (*bytes*) – Public key of the organization

- **collaborations** (*list[Collaboration]*) – List of collaborations that this organization is part of
- **results** (list[*Result*]) – List of results that are part of this organization
- **nodes** (list[*Node*]) – List of nodes that are part of this organization
- **users** (*list[User]*) – List of users that are part of this organization
- **created_tasks** (*list[Task]*) – List of tasks that are created by this organization
- **roles** (*list[Role]*) –

classmethod get_by_name(*name*)

Returns the organization with the given name.

Parameters
name (*str*) – Name of the organization

Returns
Organization with the given name if it exists, otherwise None

Return type
Organization | None

get_result_ids()

Returns a list of result ids that are part of this organization.

Returns
List of result ids

Return type
list[int]

public_key

Returns the public key of the organization.

Returns
Public key of the organization. Empty string if no public key is set.

Return type
str

class Result(***kwargs*)

Table that describes which results are available. A Result is the description of a Task as executed by a Node.

The result (and the input) is encrypted and can be only read by the intended receiver of the message.

Variables

- **input** (*str*) – Input data of the task
- **task_id** (*int*) – Id of the task that was executed
- **organization_id** (*int*) – Id of the organization that executed the task
- **result** (*str*) – Result of the task
- **assigned_at** (*datetime*) – Time when the task was assigned to the node
- **started_at** (*datetime*) – Time when the task was started
- **finished_at** (*datetime*) – Time when the task was finished
- **status** (*str*) – Status of the task

- **`log`** (*`str`*) – Log of the task
- **`task`** (`Task`) – Task that was executed
- **`organization`** (`Organization`) – Organization that executed the task
- **`ports`** (*`list[AlgorithmPort]`*) – List of ports that are part of this result

`complete`

Returns whether the algorithm run has completed or not.

Returns
True if the algorithm run has completed, False otherwise.

Return type
bool

`property node:` ***`Node`***

Returns the node that is associated with this result.

Returns
The node that is associated with this result.

Return type
model.node.Node

`class Role`(***kwargs*)

Collection of Rules

Variables

- **`name`** (*`str`*) – Name of the role
- **`description`** (*`str`*) – Description of the role
- **`organization_id`** (*`int`*) – Id of the organization this role belongs to
- **`rules`** (*`List[Rule]`*) – List of rules that belong to this role
- **`organization`** (`Organization`) – Organization this role belongs to
- **`users`** (*`List[User]`*) – List of users that belong to this role

`classmethod get_by_name`(*name*)

Get a role by its name.

Parameters
`name` (*`str`*) – Name of the role

Returns
Role with the given name or None if no role with the given name exists

Return type
Role | None

`class Rule`(***kwargs*)

Rules to determine permissions in an API endpoint.

A rule gives access to a single type of action with a given operation, scope and resource on which it acts. Note that rules are defined on startup of the server, based on permissions defined in the endpoints. You cannot edit the rules in the database.

Variables

- **`name`** (*`str`*) – Name of the rule

- **operation** (Operation) – Operation of the rule
- **scope** (Scope) – Scope of the rule
- **description** (*str*) – Description of the rule
- **roles** (*list[Role]*) – Roles that have this rule
- **users** (*list[User]*) – Users that have this rule

classmethod get_by_(*name*, *scope*, *operation*)

Get a rule by its name, scope and operation.

Parameters

- **name** (*str*) – Name of the resource on which the rule acts, e.g. 'node'
- **scope** (*str*) – Scope of the rule, e.g. 'organization'
- **operation** (*str*) – Operation of the rule, e.g. 'view'

Returns

Rule with the given name, scope and operation or None if no rule with the given name, scope and operation exists

Return type

Rule | None

class Task(***kwargs*)

Table that describes all tasks.

A task can contain multiple Results for multiple organizations. The input of the task is different for each organization (due to the encryption). Therefore the input for the task is encrypted for each organization separately. The task originates from an organization to which the results need to be encrypted, therefore the originating organization is also logged

Variables

- **name** (*str*) – Name of the task
- **description** (*str*) – Description of the task
- **image** (*str*) – Name of the docker image that needs to be executed
- **collaboration_id** (*int*) – Id of the collaboration that this task belongs to
- **run_id** (*int*) – Run id of the task
- **parent_id** (*int*) – Id of the parent task (if any)
- **database** (*str*) – Name of the database that needs to be used for this task
- **initiator_id** (*int*) – Id of the organization that created this task
- **init_user_id** (*int*) – Id of the user that created this task
- **collaboration** (*Collaboration*) – Collaboration that this task belongs to
- **parent** (*Task*) – Parent task (if any)
- **results** (list[*Result*]) – List of results that are part of this task
- **initiator** (*Organization*) – Organization that created this task
- **init_user** (*User*) – User that created this task

classmethod next_run_id()

Get the next available run id for a new task.

Returns
Next available run id

Return type
int

results_for_node(*node*)

Get all results for a given node.

Parameters
node (`model.node.Node`) – Node for which to get the results

Returns
List of results for the given node

Return type
List[*model.result.Result*]

class User(***kwargs*)

Bases: *Authenticatable*

Table to keep track of Users (persons) that can access the system.

Users always belong to an organization and can have certain rights within an organization.

Variables

- **username** (*str*) – Username of the user
- **password** (*str*) – Password of the user
- **firstname** (*str*) – First name of the user
- **lastname** (*str*) – Last name of the user
- **email** (*str*) – Email address of the user
- **organization_id** (*int*) – Foreign key to the organization to which the user belongs
- **failed_login_attempts** (*int*) – Number of failed login attempts
- **last_login_attempt** (*datetime.datetime*) – Date and time of the last login attempt
- **otp_secret** (*str*) – Secret key for one time passwords
- **organization** (*Organization*) – Organization to which the user belongs
- **roles** (list[*Role*]) – Roles that the user has
- **rules** (list[*Rule*]) – Rules that the user has
- **created_tasks** (list[*Task*]) – Tasks that the user has created

can(*resource*, *scope*, *operation*)

Check if user is allowed to execute a certain action

Parameters

- **resource** (*str*) – The resource type on which the action is to be performed
- **scope** (`Scope`) – The scope within which the user wants to perform an action
- **operation** (`Operation`) – The operation a user wants to execute

Returns
Whether or not user is allowed to execute the requested operation on the resource

Return type
bool

`check_password`(*pw*)

Check if the password is correct

Parameters
`pw` (*str*) – Password to check

Returns
Whether or not the password is correct

Return type
bool

`classmethod exists`(*field*, *value*)

Checks if user with certain key-value exists

Parameters

- **`field`** (*str*) – Name of the attribute to check
- **`value`** (*str*) – Value of the attribute to check

Returns
Whether or not user with given key-value exists

Return type
bool

`classmethod get_by_email`(*email*)

Get a user by their email

Parameters
`email` (*str*) – Email of the user

Returns
User with the given email

Return type
User

Raises
`NoResultFound` – If no user with the given email exists

`classmethod get_by_username`(*username*)

Get a user by their username

Parameters
`username` (*str*) – Username of the user

Returns
User with the given username

Return type
User

Raises
`NoResultFound` – If no user with the given username exists

`is_blocked`(*max_failed_attempts*, *inactivation_in_minutes*)

Check if user can login or if they are temporarily blocked because they entered a wrong password too often

Parameters

- **`max_failed_attempts`** (*`int`*) – Maximum number of attempts to login before temporary deactivation
- **`inactivation_minutes`** (*`int`*) – How many minutes an account is deactivated

Return type

`Tuple`[`bool`, `Optional`[`str`]]

Returns

- *bool* – Whether or not user is blocked temporarily
- *str* | *None* – Message if user is blocked, else None

`set_password`(*pw*)

Set the password of the current user. This function doesn't save the new password to the database

Parameters

`pw` (*`str`*) – The new password

Returns

If the new password fails to pass the checks, a message is returned. Else, none is returned

Return type

str | None

`classmethod username_exists`(*username*)

Checks if user with certain username exists

Parameters

`username` (*`str`*) – Username to check

Returns

Whether or not user with given username exists

Return type

bool

Database models that link resources together

The Member table is used to link organizations and collaborations together. Each line in the table represents that a certain organization is member of a certain collaboration by storing the ids of the organization and collaboration.

`Member`

alias of Table('Member', MetaData(), Column('organization_id', Integer(), ForeignKey('organization.id'), table=<Member>), Column('collaboration_id', Integer(), ForeignKey('collaboration.id'), table=<Member>), schema=None)

The Permission table defines which roles have been assigned to which users. It can contain multiple entries for the same user if they have been assigned multiple roles.

The UserPermission table defines which extra rules have been assigned to which users. Apart from roles, users may be assigned extra permissions that allow them to execute one specific action. This table is used to store those, and may contain multiple entries for the same user.

`Permission`

alias of Table('Permission', MetaData(), Column('role_id', Integer(), ForeignKey('role.id'), table=<Permission>), Column('user_id', Integer(), ForeignKey('user.id'), table=<Permission>), schema=None)

`UserPermission`

alias of Table('UserPermission', MetaData(), Column('rule_id', Integer(), ForeignKey('rule.id'), table=<UserPermission>), Column('user_id', Integer(), ForeignKey('user.id'), table=<UserPermission>), schema=None)

The role_rule_assocation table defines which rules have been assigned to which roles. Each line contains a rule_id that is a member of a certain role_id. Each role will usually have multiple rules assigned to it.

`role_rule_association`

alias of Table('role_rule_association', MetaData(), Column('role_id', Integer(), ForeignKey('role.id'), table=<role_rule_association>), Column('rule_id', Integer(), ForeignKey('rule.id'), table=<role_rule_association>), schema=None)

4.6.4.7 Mail service

`class MailService`(*app*, *mail*)

Send emails from the service email account

Parameters

- **app** (*flask.Flask*) – The vantage6 flask application
- **mail** (*flask_mail.Mail*) – An instance of the Flask mail class

`send_email`(*subject*, *sender*, *recipients*, *text_body*, *html_body*)

Send an email.

This is used for service emails, e.g. to help users reset their password.

Parameters

- **subject** (*str*) – Subject of the email
- **sender** (*str*) – Email address of the sender
- **recipients** (*List[str]*) – List of email addresses of recipients
- **text_body** (*str*) – Email body in plain text
- **html_body** (*str*) – Email body in HTML

Return type

`None`

4.6.4.8 Default roles

`get_default_roles`(*db*)

Get a list containing the default roles and their rules, so that they may be created in the database

Parameters

db – The vantage6.server.db module

Returns

A list with dictionaries that each describe one of the roles. Each role dictionary contains the following:

name: str
Name of the role

description: str
Description of the role

rules: List[int]
A list of rule id's that the role contains

Return type
List[Dict]

4.7 Developer community

As an open-source platform, we welcome anyone who would like to contribute to the vantage6 code and/or documentation. The following sections are meant to clarify our processes in development, documentation and releasing.

4.7.1 Contribute

4.7.1.1 Support questions

If you have questions, you can use

- Github discussions
- Ask us on Discord

We prefer that you ask questions via these routes rather than creating Github issues. The issue tracker is intended to address bugs, feature requests, and code changes.

4.7.1.2 Reporting issues

Issues can be posted at our Github issue page, or, if you have issues that are specific to the user interface, please post them to the UI issue page.

We distinguish between the following types of issues:

- Bug report: you encountered broken code
- Feature request: you want something to be added
- Change request: there is a something you would like to be different but it is not considered a new feature nor is something broken
- Security vulnerabilities: you found a security issue

Each issue type has its own template. Using these templates makes it easier for us to manage them.

Warning: Security vulnerabilities should not be reported in the Github issue tracker as they should not be publically visible. To see how we deal with security vulnerabilities read our policy.

See the *Security vulnerabilities* section when you want to release a security patch yourself.

We distibute the open issues in sprints and hotfixes. You can check out these boards here:

- Sprints
- Hotfixes

When a high impact bug is reported, we will put it on the hotfix board and create a patch release as soon as possible.

The sprint board tracks which issues we plan to fix in which upcoming release. Low-impact bugs, new features and changes will be scheduled into a sprint periodically. We automatically assign the label 'new' to all newly reported issues to track which issues should still be scheduled.

If you would like to fix an existing bug or create a new feature, check *Submitting patches* for more details on e.g. how to set up a local development environment and how the release process works. We prefer that you let us know you what are working on so we prevent duplicate work.

4.7.1.3 Security vulnerabilities

If you are a member of the Vantage6 Github organization, you can create an security advisory in the Security tab. See Table 4.2 on what to fill in.

If you are not a member, please reach out directly to Frank Martin and/or Bart van Beusekom, or any other project member. They can then create a security advisory for you.

Table 4.2: Advisory details

Name	Details
Ecosystem	Set to `pip`
Package name	Set to `vantage6`
Affected versions	Specify the versions (or set of verions) that are affected
Patched version	Version where the issue is addessed, you can fill this in later when the patch is released.
Severity	Determine severity score using this tool. Then use table Table 4.3 to determine the level from this score.
Common weakness enumerator (CWE)	Find the CWE (or multiple) on this website.

Table 4.3: Severity

Score	Level
0.1-3.9	Low
4.0-6.9	Medium
7.0-8.9	High
9.0-10.0	Critical

Once the advisory has been created it is possible to create a private fork from there (Look for the button `Start a temporary private fork`). This private fork should be used to solve the issue.

From the same page you should request a CVE number so we can alert dependent software projects. Github will review the request. We are not sure what this entails, but so far they approved all advisories.

4.7.1.4 Community Meetings

We host bi-monthly community meetings intended for aligning development efforts. Anyone is welcome to join although they are mainly intended for infrastructure and algorithm developers. There is an opportunity to present what your team is working on an find collaboration partners.

Community meetings are usually held on the third Thursday of the month at 11:00 AM CET on Microsoft Teams. Reach out on Discord if you want to join the community meeting.

For more information and slides from previous meetings, check our website.

4.7.1.5 Submitting patches

If there is not an open issue for what you want to submit, please open one for discussion before submitting the PR. We encourage you to reach out to us on Discord, so that we can work together to ensure your contribution is added to the repository.

The workflow below is specific to the vantage6 infrastructure repository. However, the concepts for our other repositories are the same. Then, modify the links below and ignore steps that may be irrelevant to that particular repository.

Setup your environment

- Make sure you have a Github account
- Install and configure git
- (Optional) install and configure Miniconda
- Clone the main repository locally:

  ```
  git clone https://github.com/vantage6/vantage6
  cd vantage6
  ```

- Add your fork as a remote to push your work to. Replace `{username}` with your username.

  ```
  git remote add fork https://github.com/{username}/vantage6
  ```

- Create a virtual environment to work in. For miniconda:

  ```
  conda create -n vantage6 python=3.10
  conda activate vantage6
  ```

  It is also possible to use `virtualenv` if you do not have a conda installation.

- Update pip and setuptools

  ```
  python -m pip install --upgrade pip setuptools
  ```

- Install vantage6 as development environment with the `-e` flag.

  ```
  pip install -e .
  ```

Coding

First, create a branch you can work on. Make sure you branch of the latest `main` branch:

```
git fetch origin
git checkout -b your-branch-name origin/main
```

Then you can create your bugfix, change or feature. Make sure to commit frequently. Preferably include tests that cover your changes.

Finally, push your commits to your fork on Github and create a pull request.

```
git push --set-upstream fork your-branch-name
```

Please apply the PEP8 standards to your code.

Local test setup

To test your code changes, it may be useful to create a local test setup. There are several ways of doing this.

1. Use the command `vserver-local` and `vnode-local`. This runs the application in your current activated Python environment.
2. Use the command `vserver` and `vnode` in combination with the options `--mount-src` and optionally `--image`.

- The `--mount-src` option will run your current code in the docker image. The provided path should point towards the root folder of the vantage6 repository.
- The `--image` can be used to point towards a custom build infrastructure image. Note that when your code update includes dependency upgrades you need to build a custom infrastructure image as the 'old' image does not contain these and the `--mount-src` option will only overwrite the source and not re-install dependencies.

Note: If you are using Docker Desktop (which is usually the case if you are on Windows or MacOS) and want to setup a test environment, you should use `http://host.docker.interal` for the server address in the node configuration file. You should not use `http://localhost` in that case as that points to the localhost within the docker container instead of the system-wide localhost.

Unit tests & coverage

You can execute unit tests using the `test` command in the Makefile:

```
make test
```

If you want to execute a specific unit test (e.g. the one you just created or one that is failing), you can use a command like:

```
python -m unittest tests_folder.test_filename.TestClassName.test_name
```

This command assumes you are in the directory above `tests_folder`. If you are inside the `tests_folder`, then you should remove that part.

Pull Request

Please consider first which branch you want to merge your contribution into. **Patches** are usually directly merged into `main`, but **features** are usually merged into a development branch (e.g. `dev3` for version 3) before being merged into the `main` branch.

Before the PR is merged, it should pass the following requirements:

- At least one approved review of a code owner
- All unit tests should complete
- CodeQL (vulnerability scanning) should pass
- Codacy - Code quality checks - should be OK
- Coveralls - Code coverage analysis - should not decrease

Documentation

Depending on the changes you made, you may need to add a little (or a lot) of documentation. For more information on how and where to edit the documentation, see the section *Documentation*.

Consider which documentation you need to update:

- **User documentation.** Update it if your change led to a different expierence for the end-user
- **Technical documentation.** Update it if you added new functionality. Check if your function docstrings have also been added (see last bullet below).
- **OAS (Open API Specification).** If you changed input/output for any of the API endpoints, make sure to add it to the docstrings. See *API Documenation with OAS3+* for more details.
- **Function docstrings** These should always be documented using the numpy format. Such docstrings can then be used to automatically generate parts of the technical documentation space.

4.7.2 Documentation

The vantage6 framework is documented on this website. Additionally, there is *API Documenation with OAS3+*. This documentation is shipped directly with the server instance. All of these documentation pages are described in more detail below.

4.7.2.1 How this documentation is created

The source of the documentation you are currently reading is located here, in the `docs` folder of the *vantage6* repository itself.

To build the documentation locally, there are two options. To build a static version, you can do `make html` when you are in the `docs` directory. If you want to automatically refresh the documentation whenever you make a change, you can use sphinx-autobuild. Assuming you are in the main directory of the repository, run the following commands:

```
pip install -r docs/requirements.txt
sphinx-autobuild docs docs/_build/html --watch .
```

Of course, you only have to install the requirements if you had not done so before.

Then you can access the documentation on `http://127.0.0.1:8000`. The `--watch` option makes sure that if you make changes to either the documentation text or the docstrings, the documentation pages will also be reloaded.

This documentation is automatically built and published on a commit (on certain branches, including `main`). Both Frank and Bart have access to the vantage6 project when logged into readthedocs. Here they can manage which branches are to be synced, manage the webhook used to trigger a build, and some other -less important- settings.

The files in this documentation use the `rst` format, to see the syntax view this cheatsheet.

4.7.2.2 API Documenation with OAS3+

The API documentation is hosted at the server at the `/apidocs` endpoint. This documentation is generated from the docstrings using Flasgger. The source of this documentation can be found in the docstrings of the API functions.

If you are unfamiliar with OAS3+, note that it was formerly known as Swagger.

An example of such a docsting:

```
"""Summary of the endpoint
   ---
   description: >-
       Short description on what the endpoint does, and which users have
       access or which permissions are required.

   parameters:
       - in: path
         name: id
         schema:
           type: integer
         description: some identifier
         required: true

   responses:
       200:
           description: Ok
       401:
           description: Unauthorized or missing permission

   security:
       - bearerAuth: []

   tags: ["Group"]
"""
```

4.7.3 Release

This page is intended to provide information about our release process. First, we discuss the version formatting, after which we discuss the actual creation and distribution of a release.

4.7.3.1 Version format

Semantic versioning is used: `Major.Minor.Patch.Pre[N].Post<n>`.

Major is used for releasing breaking changes. For example, when the database
model has changed, a new major version should be issued.

Minor is used for releasing new features, enhancements and other changes that
are compatible with all other components. An example is the release of a new endpoint.

Patch is used for bugfixes and other minor changes

Pre[N] is used for alpha (a), beta (b) and release candidates (rc) releases and the
build number is appended (e.g. `2.0.1b1` indicates the first beta-build of version `2.0.1`)

Post[N] is used for a rebuild where no code changes have been made, but where,
for example, a dependency has been updated and a rebuild is required.

Warning: Post releases are only used by versioning the Docker images. Code changes should never be released with a `.post[N]` version.

4.7.3.2 Create a release

To create a new release, one should go through the following steps:

- Check out the correct branch of the vantage6 repository and pull the latest version:

```
git checkout main
git pull
```

  Make sure the branch is up-to-date. **Patches** are usually directly merged into main, but for **minor** or **major** releases you usually need to execute a pull request from a development branch.

- Create a tag for the release. See *Version format* for more details on version names:

```
git tag version/x.y.z
```

- Push the tag to the remote. This will trigger the release pipeline on Github:

```
git push origin version/x.y.z
```

Note: The release process is protected and can only be executed by certain people. Reach out if you have any questions regarding this.

4.7.3.3 The release pipeline

The release pipeline executes the following steps:

1. It checks if the tag contains a valid version specification. If it does not, the process it stopped.
2. Update the version in the repository code to the version specified in the tag and commit this back to the main branch.
3. Install the dependencies and build the Python package.
4. Upload the package to PyPi.
5. Build and push the Docker image to harbor2.vantage6.ai.
6. Post a message in Discord to alert the community of the new release. This is not done if the version is a pre-release (e.g. version/x.y.0rc1).

Note: If you specify a tag with a version that already exists, the build pipeline will fail as the upload to PyPi is rejected.

The release pipeline uses a number of environment variables to, for instance, authenticate to PyPi and Discord. These variables are listed and explained in the table below.

Table 4.4: Environment variables

Secret	Description
`COMMIT_PAT`	Github Personal Access Token with commit privileges. This is linked to an individual user with admin right as the commit on the `main` needs to bypass the protections. There is unfortunately not -yet- a good solution for this.
`ADD_TO_PROJECT_PAT`	Github Personal Access Token with project management privileges. This token is used to add new issues to project boards.
`COVERALLS_TOKEN`	Token from coveralls to post the test coverage stats.
`DOCKER_TOKEN`	Token used together `DOCKER_USERNAME` to upload the container images to our https://harbor2.vantage6.ai.
`DOCKER_USERNAME`	See `DOCKER_TOKEN`.
`PYPI_TOKEN`	Token used to upload the Python packages to PyPi.
`DISCORD_RELEASE_TOKEN`	Token to post a message to the Discord community when a new release is published.

4.7.3.4 Distribute release

Nodes and servers that are already running will automatically be upgraded to the latest version of their major release when they are restarted. This happens by pulling the newly released docker image. Note that the major release is never automatically updated: for example, a node running version 2.1.0 will update to 2.1.1 or 2.2.0, but never to 3.0.0. Depending on the version of Vantage6 that is being used, there is a reserved Docker image tag for distributing the upgrades. These are the following:

Tag	Description
petronas	`3.x.x` release
harukas	`2.x.x` release
troltunga	`1.x.x` release

Docker images can be pulled manually with e.g.

```
docker pull harbor2.vantage6.ai/infrastructure/server:petronas
docker pull harbor2.vantage6.ai/infrastructure/node:3.1.0
```

4.7.3.5 User Interface release

The release process for the user interface (UI) is very similar to the release of the infrastructure detailed above. The same versioning format is used, and when you push a version tag, the automated release process is triggered.

We have semi-synchronized the version of the UI with that of the infrastructure. That is, we try to release major and minor versions at the same time. For example, if we are currently at version 3.5 and release version 3.6, we release it both for the infrastructure and for the UI. However, there may be different patch versions for both: the latest version for the infrastructure may then be 3.6.2 while the UI may still be at 3.6.

The release pipeline for the UI executes the following steps:

1. Version tag is verified (same as infrastructure).
2. Version is updated in the code (same as infrastructure).
3. Application is built.
4. Docker images are built and released to harbor2.
5. Application is pushed to our UI deployment slot (an Azure app service).

4.8 Glossary

The following is a list of definitions used in vantage6.

A

- **Autonomy:** the ability of a party to be in charge of the control and management of its own data.

C

- **Collaboration**: an agreement between two or more parties to participate in a study (i.e., to answer a research question).

D

- **Distributed learning**: see *Federated Learning*
- **Docker:** a platform that uses operating system virtualization to deliver software in packages called containers. It is worth noting that although they are often confused, Docker containers are not virtual machines.
- **Data Station**: Virtual Machine containing the vantage6-node application and a database.

F

- **FAIR data**: data that are Findable, Accessible, Interoperable, and Reusable. For more information, see the original paper.
- **Federated learning**: an approach for analyzing data that are spread across different parties. Its main idea is that parties run computations on their local data, yielding either aggregated parameters or encrypted values. These are then shared to generate a global (statistical) model. In other words, instead of bringing the data to the algorithms, federated learning brings the algorithms to the data. This way, patient-sensitive information is not disclosed. Federated learning is some times known as *distributed learning*. However, we try to avoid this term, since it can be confused with distributed computing, where different computers share their processing power to solve very complex calculations.

H

- **Heterogeneity**: the condition in which in a federated learning scenario, parties are allowed to have differences in hardware and software (i.e., operating systems).
- **Horizontally-partitioned data**: data spread across different parties where the latter have the same features of different instances (i.e., patients). See also vertically-partitioned data.

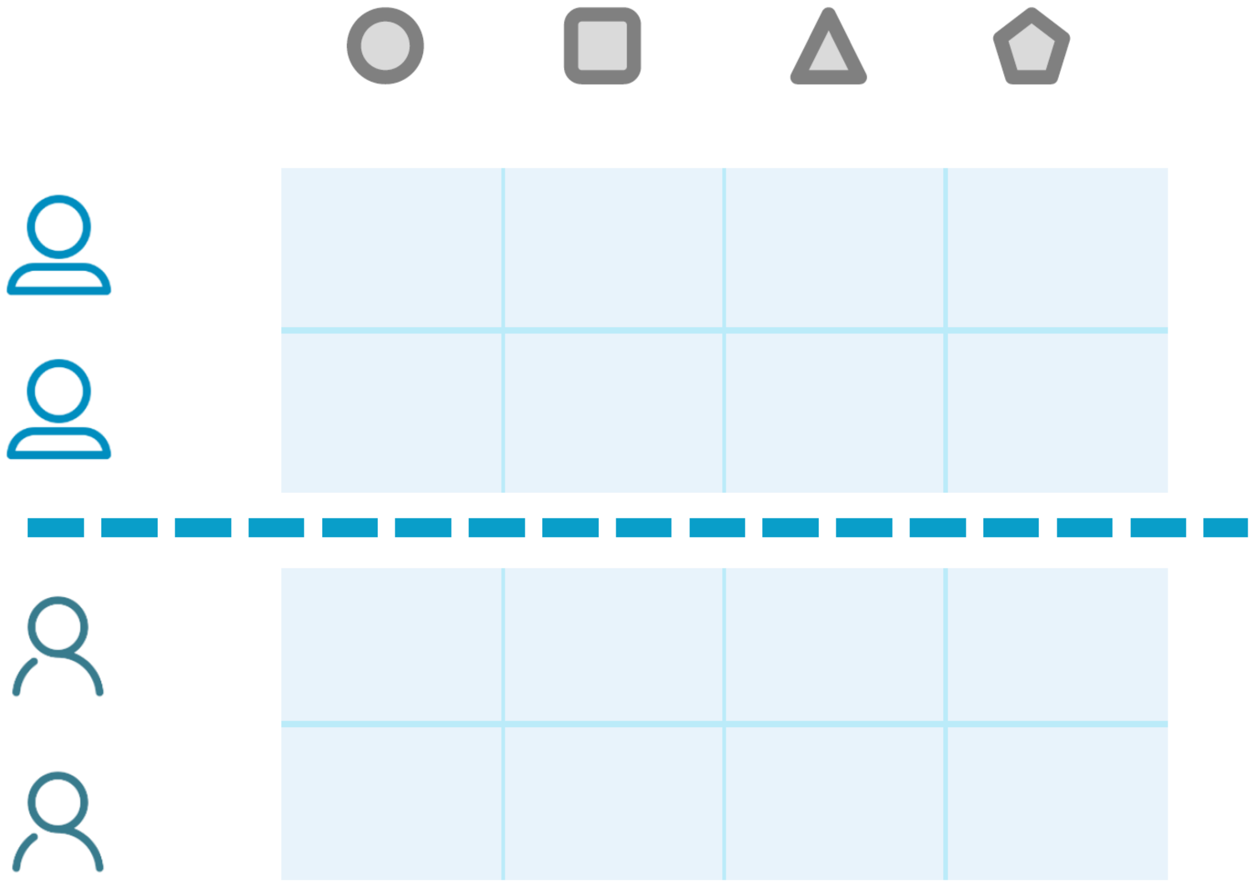


Fig. 4.11: Horizontally-partitioned data

N

- **Node**: vantage6 node application that runs at a **Data Station** which has access to the local data.

M

- **Multi-party computation**: an approach to perform analyses across different parties by performing operations on encrypted data.

P

- **Party**: an entity that takes part in one (or more) collaborations
- **Python**: a high-level general purpose programming language. It aims to help programmers write clear, logical code. vantage6 is written in Python.

S

- **Secure multi-party computation**: see *Multi-party computation*

- **Server**: Public access point of the vantage6 infrastructure. Contains at least the **vantage6-server** application but can also host the optional components: Docker registry, VPN server and RabbitMQ. In this documentation space we try to be explicit when we talk about _server_ and _vantage6-**server**_, however you might encounter _server_ where _vantage6-**server**_ should have been.

V

- **vantage6**: priVAcy preserviNg federaTed leArninG infrastructurE for Secure Insight eXchange. In short, vantage6 is an infrastructure for executing federated learning analyses. However, it can also be used as a FAIR data station and as a model repository.
- **Vertically-partitioned data**: data spread across different parties where the latter have different features of the same instances (i.e., patients). See also horizontally-partitioned data.

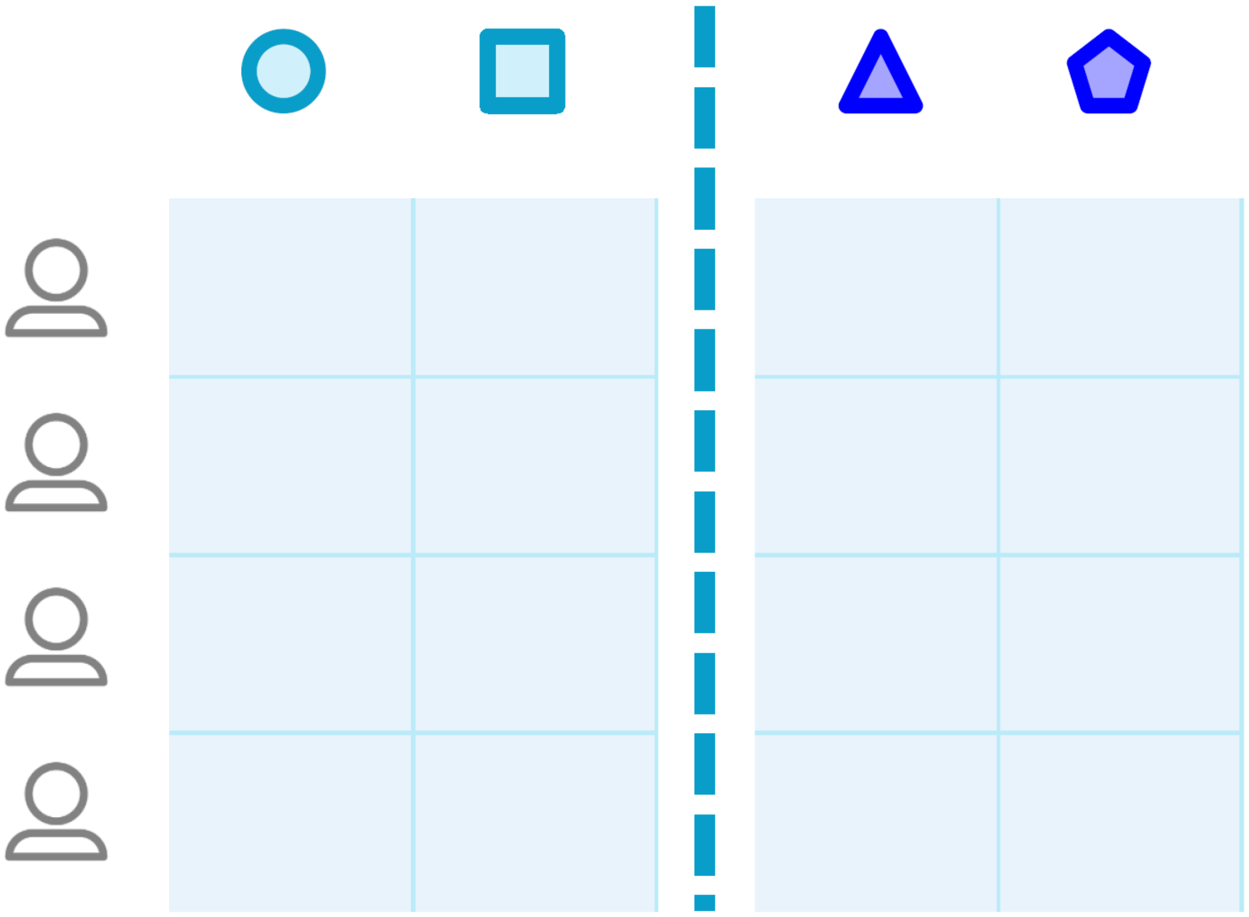


Fig. 4.12: Vertically partitioned data

4.9 Release notes

4.9.1 3.8.2

22 march 2023

- **Feature**
- Location of the server configuration file in server shell script can now be specified as an environment variable (PR#604)
- **Change**
- Changed ping/pong mechanism over socket connection between server and nodes, as it did not function properly in combination with RabbitMQ. Now, the node pushes a ping and the server periodically checks if the node is still alive (PR#593)
- **Bugfix**
- For `vnode files`, take the new formatting of the databases in the node configuration file into account (PR#600)
- Fix bugs in new algorithm client where class attributes were improperly referred to (PR#596)
- Fixed broken links in Discord notification (PR#591)

4.9.2 3.8.1

8 march 2023

- **Bugfix**
- In 3.8.0, starting RabbitMQ for horizontal scaling caused a server crash due to a missing `kombu` dependency. This dependency was wrongly removed in updating all dependencies for python 3.10 (PR#585).

4.9.3 3.8.0

8 march 2023

- **Security**
- Refresh tokens are no longer indefinitely valid (CVE#CVE-2023-23929, commit).
- It was possible to obtain usernames by brute forcing the login since v3.3.0. This was due to a change where users got to see a message their account was blocked after N failed login attempts. Now, users get an email instead if their account is blocked (CVE#CVE-2022-39228, commit).
- Assigning existing users to a different organizations was possible. This may lead to unintended access: if a user from organization A is accidentally assigned to organization B, they will retain their permissions and therefore might be able to access resources they should not be allowed to access (CVE#CVE-2023-22738, commit).
- **Feature**
- Python version upgrade to 3.10 and many dependencies are upgraded (PR#513, Issue#251).
- Added `AlgorithmClient` which will replace `ContainerClient` in v4.0. For now, the new `AlgorithmClient` can be used by specifying `use_new_client=True` in the algorithm wrapper (PR#510, Issue#493).
- It is now possible to request some of the node configuration settings, e.g. which algorithms they allow to be run (PR#523, Issue#12).

- Added `auto_wrapper` which detects the data source types and reads the data accordingly. This removes the need to rebuild every algorithm for every data source type (PR#555, Issue#553).
- New endpoint added `/vpn/algorithm/addresses` for algorithms to obtain addresses for containers that are part of the same computation task (PR#501, Issue#499).
- Added the option to allow only allow certain organization and/or users to run tasks on your node. This can be done by using the `policies` configuration option. Note that the `allowed_images` option is now nested under the `policies` option (Issue#335, PR#556)
- **Change**
- Some changes have been made to the release pipeline (PR#519, PR#488, PR#500, Issue#485).
- Removed unused script to start the shell (PR#494).
- **Bugfix**
- Algorithm containers running on the same node could not communicate with each other through the VPN. This has been fixed (PR#532, Issue#336).

4.9.4 3.7.3

22 february 2023

- **Bugfix**
- A database commit in 3.7.2 was done on the wrong variable, this has been corrected (PR#547, Issue#534).
- Delete entries in the VPN port table after the algorithm has completed (PR#548).
- Limit number of characters of the task input printed to the logs (PR#550).

4.9.5 3.7.2

20 february 2023

- **Bugfix**
- In 3.7.1, some sessions were closed, but not all. Now, sessions are also terminated in the socketIO events (PR#543, Issue#534).
- Latest versions of VPN images were not automatically downloaded by node on VPN connection startup. This has been corrected (PR#533).

4.9.6 3.7.1

16 february 2023

- **Change**
- Some changes to the release pipeline.
- **Bugfix**
- `iptables` dependency was missing in the VPN client container (PR#533 Issue#518).
- Fixed a bug that did not close Postgres DB sessions, resulting in a dead server (PR#540, Issue#534).

4.9.7 3.7.0

25 january 2023

- **Feature**
- SSH tunnels are available on the node. This allows nodes to connect to other machines over SSH, thereby greatly expanding the options to connect databases and other services to the node, which before could only be made available to the algorithms if they were running on the same machine as the node (PR#461, Issue#162).
- For two-factor authentication, the information given to the authenticator app has been updated to include a clearer description of the server and username (PR#483, Issue#405).
- Added the option to run an algorithm without passing data to it using the CSV wrapper (PR#465)
- In the UI, when users are about to create a task, they will now be shown which nodes relevant to the task are offline (PR#97, Issue#96).
- **Change**
- The `docker` dependency is updated, so that `docker.pull()` now pulls the *default* tag if no tag is specified, instead of all tags (PR#481, Issue#473).
- If a node cannot authenticate to the server because the server cannot be found, the user now gets a clearer error message(PR#480, Issue#460).
- The default role 'Organization admin' has been updated: it now allows to create nodes for their own organization (PR#489).
- The release pipeline has been updated to 1) release to PyPi as last step (since that is irreversible), 2) create release branches, 3) improve the check on the version tag, and 4) update some soon-to-be-deprecated commands (PR#488.
- Not all nodes are alerted any more when a node comes online (PR#490).
- Added instructions to the UI on how to report bugs (PR#100, Issue#57).
- **Bugfix**
- Newer images were not automatically pulled from harbor on node or server startup. This has been fixed (PR#482, Issue#471).

4.9.8 3.6.1

12 january 2023

- **Feature**
- Algorithm containers can be killed from the client. This can be done for a specific task or it possible to kill all tasks running at a specific node (PR#417, Issue#167).
- Added a `status` field for an algorithm, that tracks if an algorithm has yet to start, is started, has finished, or has failed. In the latter case, it also indicates how/when the algorithm failed (PR#417).
- The UI has been connected to the socket, and gives messages about node and task status changes (UI PR#84, UI Issue #73). There are also new permissions for socket events on the server to authorize users to see events from their (or all) collaborations (PR#417).
- It is now possible to create tasks in the UI (UI version >3.6.0). Note that all tasks are then JSON serialized and you will not be able to run tasks in an encrypted collaboration (as that would require uploading a private key to a browser) (*PR#90*).

> **Warning:** If you want to run the UI Docker image, note that from this version onwards, you have to define the `SERVER_URL` and `API_PATH` environment variables (compared to just a `API_URL` before).

- There is a new multi-database wrapper that will forward a dictionary of all node databases and their paths to the algorithm. This allows you to use multiple databases in a single algorithm easily. (PR#424, Issue #398).
- New rules are now assigned automatically to the default root role. This ensures that rules that are added in a new version are assigned to system administrators, instead of them having to change the database (PR#456, Issue #442).
- There is a new command `vnode set-api-key` that facilitates putting your API key into the node configuration file (PR#428, Issue #259).
- Logging in the Python client has been improved: instead of all or nothing, log level is now settable to one of debug, info, warn, error, critical (PR#453, Issue #340).
- When there is an error in the VPN server configuration, the user receives clearer error messages than before (PR#444, Issue #278).
- **Change**
- The node status (online/offline) is now checked periodically over the socket connection via a ping/pong construction. This is an improvement over the older version where a node's status was changed only when it connected or disconnected (PR#450, Issue #40).

> **Warning:** If a server upgrades to 3.6.1, the nodes should also be upgraded. Otherwise, the node status will be incorrect and the logs will show errors periodically with each attempted ping/pong.

- It is no longer possible for any user to change the username of another user, as this would be confusing for that user when logging in (PR#433, Issue #396).
- The server has shorter log messages when someone calls a non-existing route. The resulting 404 exception is no longer logged (PR#452, Issue #393).
- Removed old, unused scripts to start a node (PR#464).
- **Bugfix**
- Node was unable to pull images from Docker Hub; this has been corrected. (PR#432, Issue#422).
- File-based database extensions were always converted to `.csv` when they were mounted to a node. Now, files keep their original file extensions (PR#426, Issue #397).
- When a node configuration defined a wrong VPN subnet, the VPN connection didn't work but this was not detected until VPN was used. Now, the user is alerted immediately and VPN is turned off (PR#444).
- If a user tries to write a node or server config file to a non-existing directory, they are now getting a clear error message instead of an incorrect one (PR#455, Issue #1)
- There was a circular import in the infrastructure code, which has now been resolved (PR#451, Issue #53).
- In PATCH `/user`, it was not possible to set some fields (e.g. `firstname`) to an empty string if there was a value before. (PR#439, Issue #334).

Note: Release 3.6.0 was skipped due to an issue in the release process.

4.9.9 3.5.2

30 november 2022

- **Bugfix**
- Fix for automatic addition of column. This failed in some SQL dialects because reserved keywords (i.e. 'user' for PostgresQL) were not escaped (PR#415)
- Correct installation order for uWSGI in node and server docker file (PR#414)

4.9.10 3.5.1

30 november 2022

- **Bugfix**
- Backwards compatibility for which organization initiated a task between v3.0-3.4 and v3.5 (PR#412)
- Fixed VPN client container. Entry script was not executable in Github pipelines (PR#413)

4.9.11 3.5.0

30 november 2022

Warning: When upgrading to 3.5.0, you might need to add the **otp_secret** column to the **user** table manually in the database. This may be avoided by upgrading to 3.5.2.

- **Feature**
- Multi-factor authentication via TOTP has been added. Admins can enforce that all users enable MFA (PR#376, Issue#355).
- You can now request all tasks assigned by a given user (PR#326, Issue#43).
- The server support email is now settable in the configuration file, used to be fixed at `support@vantage6.ai` (PR#330, Issue#319).
- When pickles are used, more task info is shown in the node logs (PR#366, Issue#171).
- **Change**
- The `harbor2.vantag6.ai/infrastructure/algorithm-base:[TAG]` is tagged with the vantage6-client version that is already in the image (PR#389, Issue#233).
- The infrastructure base image has been updated to improve build time (PR#406, Issue#250).

4.9.12 3.4.2

3 november 2022

- **Bugfix**
- Fixed a bug in the local proxy server which made algorithm containers crash in case the *client.create_new_task* method was used (PR#382).
- Fixed a bug where the node crashed when a non existing image was sent in a task (PR#375).

4.9.13 3.4.0 & 3.4.1

25 oktober 2022

- **Feature**
- Add columns to the SQL database on startup (PR#365, ISSUE#364). This simpifies the upgrading proces when a new column is added in the new release, as you do no longer need to manually add columns. When downgrading the columns will **not** be deleted.
- Docker wrapper for Parquet files (PR#361, ISSUE#337). Parquet provides a way to store tabular data with the datatypes included which is an advantage over CSV.
- When the node starts, or when the client is verbose initialized a banner to cite the vantage6 project is added (PR#359, ISSUE#356).
- In the client a waiting for results method is added (PR#325, ISSUE#8). Which allows you to automatically poll for results by using `client.wait_for_results(...)`, for more info see `help(client.wait_for_results)`.
- Added Github releases (PR#358, ISSUE#357).
- Added option to filter GET `/role` by user id in the Python client (PR#328, ISSUE#213). E.g.: `client.role.list(user=...)`.
- In release process, build and release images for both ARM and x86 architecture.
- **Change**
- Unused code removed from the Makefile (PR#324, ISSUE#284).
- Pandas version is frozen to version 1.3.5 (PR#363 , ISSUE#266).
- **Bugfix**
- Improve checks for non-existing resources in unittests (PR#320, ISSUE#265). Flask did not support negative ints, so the tests passed due to another 404 response.
- `client.node.list` does no longer filter by offline nodes (PR#321, ISSUE#279).

Note: 3.4.1 is a rebuild from 3.4.0 in which the all dependencies are fixed, as the build led to a broken server image.

4.9.14 3.3.7

- **Bugfix**
- The function `client.util.change_my_password()` was updated (Issue #333)

4.9.15 3.3.6

- **Bugfix**
- Temporary fix for a bug that prevents the master container from creating tasks in an encrypted collaboration. This temporary fix disables the parallel encryption module in the local proxy. This functionality will be restored in a future release.

Note: This version is also the first version where the User Interface is available in the right version. From this point onwards, the user interface changes will also be part of the release notes.

4.9.16 3.3.5

- **Feature**
- The release pipeline has been expanded to automatically push new Docker images of node/server to the harbor2 service.
- **Bugfix**
- The VPN IP address for a node was not saved by the server using the PATCH `/node` endpoint, while this functionality is required to use the VPN

Note: Note that 3.3.4 was only released on PyPi and that version is identical to 3.3.5. That version was otherwise skipped due to a temporary mistake in the release pipeline.

4.9.17 3.3.3

- **Bugfix**
- Token refresh was broken for both users and nodes. (Issue#306, PR#307)
- Local proxy encrpytion was broken. This prefented algorithms from creating sub tasks when encryption was enabled. (Issue#305, PR#308)

4.9.18 3.3.2

- **Bugfix**
- `vpn_client_image` and `network_config_image` are settable through the node configuration file. (PR#301, Issue#294)
- The option `--all` from `vnode stop` did not stop the node gracefully. This has been fixed. It is possible to force the nodes to quit by using the `--force` flag. (PR#300, Issue#298)
- Nodes using a slow internet connection (high ping) had issues with connecting to the websocket channel. (PR#299, Issue#297)

4.9.19 3.3.1

- **Bugfix**
- Fixed faulty error status codes from the `/collaboration` endpoint (PR#287).
- *Default* roles are always returned from the `/role` endpoint. This fixes the error when a user was assigned a *default* role but could not reach anything (as it could not view its own role) (PR#286).
- Performance upgrade in the `/organization` endpoint. This caused long delays when retrieving organization information when the organization has many tasks (PR#288).
- Organization admins are no longer allowed to create and delete nodes as these should be managed at collaboration level. Therefore, the collaboration admin rules have been extended to include create and delete nodes rules (PR#289).
- Fixed some issues that made `3.3.0` incompatible with `3.3.1` (Issue#285).

4.9.20 3.3.0

- **Feature**
- Login requirements have been updated. Passwords are now required to have sufficient complexity (8+ characters, and at least 1 uppercase, 1 lowercase, 1 digit, 1 special character). Also, after 5 failed login attempts, a user account is blocked for 15 minutes (these defaults can be changed in a server config file).
- Added endpoint `/password/change` to allow users to change their password using their current password as authentication. It is no longer possible to change passwords via `client.user.update()` or via a PATCH `/user/{id}` request.
- Added the default roles 'viewer', 'researcher', 'organization admin' and 'collaboration admin' to newly created servers. These roles may be assigned to users of any organization, and should help users with proper permission assignment.
- Added option to filter get all roles for a specific user id in the GET `/role` endpoint.
- RabbitMQ has support for multiple servers when using `vserver start`. It already had support for multiple servers when deploying via a Docker compose file.
- When exiting server logs or node logs with Ctrl+C, there is now an additional message alerting the user that the server/node is still running in the background and how they may stop them.
- **Change**
- Node proxy server has been updated
- Updated PyJWT and related dependencies for improved JWT security.
- When nodes are trying to use a wrong API key to authenticate, they now receive a clear message in the node logs and the node exits immediately.
- When using `vserver import`, API keys must now be provided for the nodes you create.
- Moved all swagger API docs from YAML files into the code. Also, corrected errors in them.
- API keys are created with UUID4 instead of UUID1. This prevents that UUIDs created milliseconds apart are not too similar.
- Rules for users to edit tasks were never used and have therefore been deleted.
- **Bugfix**
- In the Python client, `client.organization.list()` now shows pagination metadata by default, which is consistent all other `list()` statements.
- When not providing an API key in `vnode new`, there used to be an unclear error message. Now, we allow specifying an API key later and provide a clearer error message for any other keys with inadequate values.
- It is now possible to provide a name when creating a name, both via the Python client as via the server.
- A GET `/role` request crashed if parameter `organization_id` was defined but not `include_root`. This has been resolved.
- Users received an 'unexpected error' when performing a GET `/collaboration?organization_id=<id>` request and they didn't have global collaboration view permission. This was fixed.
- GET `/role/<id>` didn't give an error if a role didn't exist. Now it does.

4.9.21 3.2.0

- **Feature**
- Horizontal scaling for the vantage6-server instance by adding support for RabbitMQ.
- It is now possible to connect other docker containers to the private algorithm network. This enables you to attach services to the algorithm network using the `docker_services` setting.
- Many additional select and filter options on API endpoints, see swagger docs endpoint (`/apidocs`). The new options have also been added to the Python client.
- Users are now always able to view their own data
- Usernames can be changed though the API
- **Bugfix**
- (Confusing) SQL errors are no longer returned from the API.
- Clearer error message when an organization has multiple nodes for a single collaboration.
- Node no longer tries to connect to the VPN if it has no `vpn_subnet` setting in its configuration file.
- Fix the VPN configuration file renewal
- Superusers are no longer able to post tasks to collaborations its organization does not participate in. Note that superusers were never able to view the results of such tasks.
- It is no longer possible to post tasks to organization which do not have a registered node attach to the collaboration.
- The `vnode create-private-key` command no longer crashes if the ssh directory does not exist.
- The client no longer logs the password
- The version of the `alpine` docker image (that is used to set up algorithm runs with VPN) was fixed. This prevents that many versions of this image are downloaded by the node.
- Improved reading of username and password from docker registry, which can be capitalized differently depending on the docker version.
- Fix error with multiple-database feature, where default is now used if specific database is not found

4.9.22 3.1.0

- **Feature**
- Algorithm-to-algorithm communication can now take place over multiple ports, which the algorithm developer can specify in the Dockerfile. Labels can be assigned to each port, facilitating communication over multiple channels.
- Multi-database support for nodes. It is now also possible to assign multiple data sources to a single node in Petronas; this was already available in Harukas 2.2.0. The user can request a specific data source by supplying the *database* argument when creating a task.
- The CLI commands `vserver new` and `vnode new` have been extended to facilitate configuration of the VPN server.
- Filter options for the client have been extended.
- Roles can no longer be used across organizations (except for roles in the default organization)
- Added `vnode remove` command to uninstall a node. The command removes the resources attached to a node installation (configuration files, log files, docker volumes etc).

- Added option to specify configuration file path when running `vnode create-private-key`.
- **Bugfix**
- Fixed swagger docs
- Improved error message if docker is not running when a node is started
- Improved error message for `vserver version` and `vnode version` if no servers or nodes are running
- Patching user failed if users had zero roles - this has been fixed.
- Creating roles was not possible for a user who had permission to create roles only for their own organization - this has been corrected.

4.9.23 3.0.0

- **Feature**
- Direct algorithm-to-algorithm communication has been added. Via a VPN connection, algorithms can exchange information with one another.
- Pagination is added. Metadata is provided in the headers by default. It is also possible to include them in the output body by supplying an additional parameter`include=metadata`. Parameters `page` and `per_page` can be used to paginate. The following endpoints are enabled:
  - GET `/result`
  - GET `/collaboration`
  - GET `/collaboration/{id}/organization`
  - GET `/collaboration/{id}/node`
  - GET `/collaboration/{id}/task`
  - GET `/organization`
  - GET `/role`
  - GET `/role/{id}/rule`
  - GET `/rule`
  - GET `/task`
  - GET `/task/{id}/result`
  - GET `/node`
- API keys are encrypted in the database
- Users cannot shrink their own permissions by accident
- Give node permission to update public key
- Dependency updates
- **Bugfix**
- Fixed database connection issues
- Don't allow users to be assigned to non-existing organizations by root
- Fix node status when node is stopped and immediately started up
- Check if node names are allowed docker names

4.9.24 2.3.0 - 2.3.4

- **Feature**
- Allows for horizontal scaling of the server instance by adding support for RabbitMQ. Note that this has not been released for version 3(!)
- **Bugfix**
- Performance improvements on the `/organization` endpoint

4.9.25 2.2.0

- **Feature**
- Multi-database support for nodes. It is now possible to assign multiple data sources to a single node. The user can request a specific data source by supplying the *database* argument when creating a task.
- The mailserver now supports TLS and SSL options
- **Bugfix**
- Nodes are now disconnected more gracefully. This fixes the issue that nodes appear offline while they are in fact online
- Fixed a bug that prevented deleting a node from the collaboration
- A role is now allowed to have zero rules
- Some http error messages have improved
- Organization fields can now be set to an empty string

4.9.26 2.1.2 & 2.1.3

- **Bugfix**
- Changes to the way the application interacts with the database. Solves the issue of unexpected disconnects from the DB and thereby freezing the application.

4.9.27 2.1.1

- **Bugfix**
- Updating the country field in an organization works again\
- The `client.result.list(...)` broke when it was not able to deserialize one of the in- or outputs.

4.9.28 2.1.0

- **Feature**
- Custom algorithm environment variables can be set using the `algorithm_env` key in the configuration file. See this Github issue.
- Support for non-file-based databases on the node. See this Github issue.
- Added flag `--attach` to the `vserver start` and `vnode start` command. This directly attaches the log to the console.
- Auto updating the node and server instance is now limited to the major version. See this Github issue.
  - e.g. if you've installed the Trolltunga version of the CLI you will always get the Trolltunga version of the node and server.
  - Infrastructure images are now tagged using their version major. (e.g. `trolltunga` or `harukas`)
  - It is still possible to use intermediate versions by specifying the `--image` option when starting the node or server. (e.g. `vserver start --image harbor.vantage6.ai/infrastructure/server:2.0.0.post1`)
- **Bugfix**
- Fixed issue where node crashed if the database did not exist on startup. See this Github issue.

4.9.29 2.0.0.post1

- **Bugfix**
- Fixed a bug that prevented the usage of secured registry algorithms

4.9.30 2.0.0

- **Feature**
- Role/rule based access control
  - Roles consist of a bundle of rules. Rules profided access to certain API endpoints at the server.
  - By default 3 roles are created: 1) Container, 2) Node, 3) Root. The root role is assigned to the root user on the first run. The root user can assign rules and roles from there.
- Major update on the *python*-client. The client also contains management tools for the server (i.e. to creating users, organizations and managing permissions. The client can be imported from `from vantage6.client import Client` .
- You can use the agrument `verbose` on the client to output status messages. This is usefull for example when working with Jupyter notebooks.
- Added CLI `vserver version` , `vnode version` , `vserver-local version` and `vnode-local version` commands to report the version of the node or server they are running
- The logging contains more information about the current setup, and refers to this documentation and our Discourd channel
- **Bugfix**
- Issue with the DB connection. Session management is updated. Error still occurs from time to time but can be reset by using the endpoint `/health/fix` . This will be patched in a newer version.

4.9.31 1.2.3

- **Feature**
- The node is now compatible with the Harbor v2.0 API

4.9.32 1.2.2

- **Bug fixes**
- Fixed a bug that ignored the `--system` flag from `vnode start`
- Logging output muted when the `--config` option is used in `vnode start`
- Fixed config folder mounting point when the option `--config` option is used in `vnode start`

4.9.33 1.2.1

- **Bug fixes**
- starting the server for the first time resulted in a crash as the root user was not supplied with an email address.
- Algorithm containers could still access the internet through their host. This has been patched.

4.9.34 1.2.0

- **Features**
- Cross language serialization. Enabling algorithm developers to write algorithms that are not language dependent.
- Reset password is added to the API. For this purpose two endpoints have been added: `/recover/lost`and `recover/reset` . The server config file needs to extended to be connected to a mail-server in order to make this work.
- User table in the database is extended to contain an email address which is mandatory.
- **Bug fixes**
- Collaboration name needs to be unique
- API consistency and bug fixes:
  - GET `organization` was missing domain key
  - PATCH `/organization` could not patch domain
  - GET `/collaboration/{id}/node` has been made consistent with `/node`
  - GET `/collaboration/{id}/organization` has been made consistent with `/organization`
  - PATCH `/user` root-user was not able to update users
  - DELETE `/user` root-user was not able to delete users
  - GET `/task` null values are now consistent: `[]` is replaced by `null`
  - POST, PATCH, DELETE `/node` root-user was not able to perform these actions
  - GET `/node/{id}/task` output is made consistent with the
- **other**
- `questionairy` dependency is updated to 1.5.2

- `vantage6-toolkit` repository has been merged with the `vantage6-client` as they were very tight coupled.

4.9.35 1.1.0

- **Features**
- new command `vnode clean` to clean up temporary docker volumes that are no longer used
- Version of the individual packages are printed in the console on startup
- Custom task and log directories can be set in the configuration file
- Improved **CLI** messages
- Docker images are only pulled if the remote version is newer. This applies both to the node/server image and the algorithm images
- Client class names have been simplified (`UserClientProtocol` -> `Client`)
- **Bug fixes**
- Removed defective websocket watchdog. There still might be disconnection issues from time to time.

4.9.36 1.0.0

- **Updated Command Line Interface (CLI)**
- The commands `vnode list` , `vnode start` and the new command`vnode attach` are aimed to work with multiple nodes at a single machine.
- System and user-directories can be used to store configurations by using the `--user/--system` options. The node stores them by default at user level, and the server at system level.
- Current status (online/offline) of the nodes can be seen using `vnode list` , which also reports which environments are available per configuration.
- Developer container has been added which can inject the container with the source. `vnode start --develop [source]`. Note that this Docker image needs to be build in advance from the `development.Dockerfile` and tag `devcon`.
- `vnode config_file` has been replaced by `vnode files` which not only outputs the config file location but also the database and log file location.
- **New database model**
- Improved relations between models, and with that, an update of the Python API.
- Input for the tasks is now stored in the result table. This was required as the input is encrypted individually for each organization (end-to-end encryption (E2EE) between organizations).
- The `Organization` model has been extended with the `public_key` (String) field. This field contains the public key from each organization, which is used by the E2EE module.
- The `Collaboration` model has been extended with the `encrypted` (Boolean) field which keeps track if all messages (tasks, results) need to be E2EE for this specific collaboration.
- The `Task` keeps track of the initiator (organization) of the organization. This is required to encrypt the results for the initiator.
- **End to end encryption**
- All messages between all organizations are by default be encrypted.

- Each node requires the private key of the organization as it needs to be able to decrypt incoming messages. The private key should be specified in the configuration file using the `private_key` label.
- In case no private key is specified, the node generates a new key an uploads the public key to the server.
- If a node starts (using `vnode start`), it always checks if the `public_key` on the server matches the private key the node is currently using.
- In case your organization has multiple nodes running they should all point to the same private key.
- Users have to encrypt the input and decrypt the output, which can be simplified by using our client `vantage6.client.Client` __ for Python __ or `vtg::Client` __ for R.
- Algorithms are not concerned about encryption as this is handled at node level.
- **Algorithm container isolation**
- Containers have no longer an internet connection, but are connected to a private docker network.
- Master containers can access the central server through a local proxy server which is both connected to the private docker network as the outside world. This proxy server also takes care of the encryption of the messages from the algorithms for the intended receiving organization.
- In case a single machine hosts multiple nodes, each node is attached to its own private Docker network.
- **Temporary Volumes**
- Each algorithm mounts temporary volume, which is linked to the node and the `run_id` of the task
- The mounting target is specified in an environment variable `TEMPORARY_FOLDER`. The algorithm can write anything to this directory.
- These volumes need to be cleaned manually. (`docker rm VOLUME_NAME`)
- Successive algorithms only have access to the volume if they share the same `run_id` . Each time a **user** creates a task, a new `run_id` is issued. If you need to share information between containers, you need to do this through a master container. If a master container creates a task, all slave tasks will obtain the same `run_id`.
- **RESTful API**
- **All RESTful API output is HATEOS formatted.**
  (wiki)
- **Local Proxy Server**
- Algorithm containers no longer receive an internet connection. They can only communicate with the central server through a local proxy service.
- It handles encryption for certain endpoints (i.e. `/task`, the input or `/result` the results)
- **Dockerized the Node**
- All node code is run from a Docker container. Build versions can be found at our Docker repository: `harbor.distributedlearning.ai/infrastructure/node` . Specific version can be pulled using tags.
- For each running node, a Docker volume is created in which the data, input and output is stored. The name of the Docker volume is: `vantage-NODE_NAME-vol` . This volume is shared with all incoming algorithm containers.
- Each node is attached to the public network and a private network: `vantage-NODE_NAME-net`.

4.10 Partners

Our community is open to everyone. The following people and organizations made a significant contribution to the design and implementation of vantage6.

- Anja van Gestel
- Bart van Beusekom
- Frank Martin
- Hasan Alradhi
- Melle Sieswerda
- Gijs Geleijnse

- Djura Smits
- Lourens Veen

- Johan van Soest

Would you like to contribute? Check out *how to contribute!* Find and chat with us via the Discord chat!

PYTHON MODULE INDEX

INDEX