Sovr

Release 0.1

Reza Hasanzadeh

Dec 24, 2022

CONTENTS

Sovr Protocol integrates the *Golem network*'s compute prowess into the *FairOS-DFS* and the *Swarm network*'s storage facilities. The whole idea revolves around the concept of *compute pods*. A compute pod is simply a set of files that define how a *Golem* task should run, what are the inputs, outputs, and finally where the logs are going to be held. All these files are finally stored in decentralized storage of the *FaiOS-DFS* and *Swarm*. The tool that is in charge of persisting, sharing, and running compute pods is called the **Sovr CLI**.

To get your feet wet, check out the *Usage* section for further information about how to get started with the CLI and necessary prerequisites to get it set up. To deep dive into the builiding blocks of the **Sovr Protocol**, please take a look at the *Compute Pods* section.

Note: This project is under active development.

CHAPTER

ONE

CONTENTS

1.1 Usage

1.1.1 Prerequisites

There are few steps that need to be done before using the CLI:

- **Swarm**

  To setup the Swarm node, you need to consult https://docs.ethswarm.org/docs. Once your node is up, fund your wallet(you can request funds from the Swarm discord), deploy a chequebook, and bingo!

- **FairOS-DFS**

  To setup the FairOS-DFS tools, You need two files: `dfs-linux...` and `dfs-cli-linux...` which can be downloaded from https://github.com/fairDataSociety/fairOS-dfs. Please consult https://docs.fairos.fairdatasociety.org/docs/fairOS-dfs/introduction for further information on customizing FairOS-DFS. FairOS-DFS provides abstractions that facilitate the storage of fairly complex objects on Swarm so it depends on a running Swarm node.

- **Golem**

  Golem is needed to run compute pods, so please consult https://handbook.golem.network/requestor-tutorials/flash-tutorial-of-requestor-development for an in-depth setup. The following provides a compressed version of the official documentation.

  1. Obtain Yagna, the dispatching engine of the Golem

     ```
     curl -sSf https://join.golem.network/as-requestor | bash -
     ```

  2. Make sure you have installed Yagna

     ```
     yagna --version
     gftp --version
     ```

  3. Run the Yagna daemon

     ```
     yagna service run
     ```

  4. Open a new terminal tab/window, and create a new wallet named *requestor*

     ```
     yagna app-key create requestor
     ```

  5. Get to know your wallet/keys with

```
yagna app-key list
```

Which outputs something similar to the following table

```
┌─────────────────────────────────────────────────────────────────────────────────────────
│  name         │  key                                │  id                              ␣
↪                          │  role       │  created                               │
├─────────────────────────────────────────────────────────────────────────────────────────
│  requestor  │  1c8c96a66950905baeb48014d7369ac6  │ ␣
↪0xb2e10dacce97f932f1d03757ff33b443f17a1c5f  │  manager  │  2022-10-
↪06T13:45:04.897349774  │
└─────────────────────────────────────────────────────────────────────────────────────────
```

Copy the key column's value.

6. Fund your wallet with testnet tokens

```
yagna payment fund
yagna payment status
```

7. Enable the Yagna daemon as a requestor

```
yagna payment init --sender
```

8. Export the key for the requestor wallet as an environment variable

```
export YAGNA_APPKEY=1c8c96a66950905baeb48014d7369ac6
```

Just remember that you need to redo this step every time you start the Yagna server.

9. Initialize a virtual environment for python 3 and install necessary libraries.

```
python3 -m venv ~/.envs/venv
source ~/.envs/venv/bin/activate

pip install -U pip
pip install yapapi requests-toolbelt
```

And done!

Now that you have a working environment for Swarm, FairOS-DFS, and Golem, it is time to start using the Sovr CLI.

1.1.2 CLI

Before use Sovr CLI, you need to have a virtual environment set up and activated with required libraries, make sure *requests-toolbelts* and *yapapi* are installed. To get the Sovr CLI, fork it from Github:

```
git clone https://github.com/rezahsnz/sovr.git
```

If you needed any help, just invoke Sovr CLI with `--help` argument:

```
usage: cli.py [-h] [--recipe RECIPE] [--persist-self]
              [--persist | --fork FORK | --run | --import-pod IMPORT_POD | --list-pods | --
↪generate-pod-registry]
```

Sovr command line interface

optional arguments:

-h, --help	show this help message and exit
--recipe RECIPE	specify a recipe file
--persist-self	Persist the CLI itself and make it public. Caution: remove any credentials(password files, ...) before proceeding.
--persist	Persist pod to dfs
--fork FORK	Fork a public pod, a reference key is expected
--run	Run the pod/task
--import-pod IMPORT_POD	Imports a pod to local filesystem, a pod name is expected
--list-pods	List all pods
--generate-pod-registry	Generate a new pod registry by looking into all pods

1.2 Compute Pods

1.2.1 Overview

Golem is a peer-to-peer compute marketplace where requestors(those who need to get some compute done) and providers(those who provide their computer in exchange for $) are matched together in a decentralised way. A typical program on Golem is usually a set of `payload(inputs)`, `scripts`, `logs`, and `outputs` with *scripts* controlling the logic of the whole running session. Note however that this is a loose definition of a Golem program as there is no such pre-defined structure and developers are free to layout their program files in any manner the wish and the layout Sovr uses is only for the sake of tidiness.

A compute pod is a logical directory structure that contains directories and files representing a program that can be run on Golem. To differentiate a compute pod from typical pods, recall that users might have other pods in their wallet too, a `.recipe` file is stored at the root of the compute pods, e.g. `/segmentation-job.recipe`. A recipe is a json file with the following look and feel:

```
{
  "name": "blender",
  "description": "Blender requestor pod that renders a .blend file.",
  "version": "1.0",
  "author": "reza",
  "public": true,
  "golem": {
    "exec": "python3 script/blender.py",
    "script": "script",
    "payload": "payload",
    "output": "output",
    "log": "logs"
  }
}
```

- `name`

This property identifies the compute pod within the FairOS-DFS and thus should be unique for every compute pod.

- `description`

  A description of the objective of the compute pod

- `author`

  The author of the compute pod

- `version`

  Version number

- `public`

  Whether the compute pod should be shared with others

- `golem`

  Defines a Golem program

  - `exec`

    The command that starts a Golem session run

  - `script`

    The logic of the Golem program is out here

  - `payload`

    Defines the payload(input) of the Golem program

  - `output`

    The output of the Golem program

  - `log`

    The logs of any Golem session runs

1.2.2 What is a Compute Pod?

A compute pod is the main building block of the Sovr Protocol that provide an easy to use scheme to manage Golem compute session. Compute pods can be run, persisted, shared, and forked by users in a decentralized way. This allows them to be viewed as portable compute objects. Compute pods might experience various independent stages during their lifetime:

- *Running*

  A running compute pod is simply a Golem program. An example of running a compute pod via Sovr CLI is shown below:

```
python src/cli.py --recipe src/templates/pods/blender/recipe.py --run
```

- *Persisting*

  A compute pod can be saved(persisted) to the FairOS-DFS with the `--persist` option of Sovr CLI. When persisting, if the `public` property of the recipe is set to **True**, the compute pod is also shared to the outside world. An example of persisting is shown below:

```
python src/cli.py --recipe src/templates/pods/blender/recipe.py --persist
```

- *Forking*

  Forking is the opposite of persisting and as its name implies, brings a public pod to the user, allowing her to build upon other people's work. `--fork` option is employed to fork compute pods. an example of forking is shown below:

```
python src/cli.py --fork 2cf98c3...23ee9a
```

Besides working with compute pods, Sovr CLI provides means to maintain the overall status of itself and compute pods. `--persist-self` for example, persists a copy of Sovr CLI(the `src/` directory) on Swarm and shares it as a measure of redundancy. Another set of options revolve around the maintenance of compute pods with `--list-pods` providing a list of current compute pods and `--generate-pod-registry` creating a registry of compute pods as users could have several other pods too and it is important to track compute pods down.

Payload and output

The notion of *payload* is very important for a compute pod as it provides means to communicate with other compute pods. A recipe defines what payload the compute pod expects. There are two types of payloads: *internal*, and *external*. An internal payload is simply the set of local files stored in the directory defined by the `payload` property while an external payload is a set of references to public pods. The following snippet shows an external payload:

```
"golem": {
  .
  .
  .
  "payload": [
      {
        "ref": "ej38b1...",
        "data": "/data.zip"
      },
      {
        "ref": "1a20fd...",
        "data": "/jake/lime.zip"
      },
      .
      .
      .
    ],
  },
  .
  .
  .
},
```

As you can see the payload requires external resource stored on public pods that need to be forked before a compute pod could use them. This is taken care of by the Sovr CLI when running a compute pod and stored in the `payload/external` direcory. Once a compute pod is ready to be persisted, the recipe could ask for its output to be shared. An example of a output sharing is given below:

```
"golem": {
    .
    .
    .
    "output": {
```

(continues on next page)

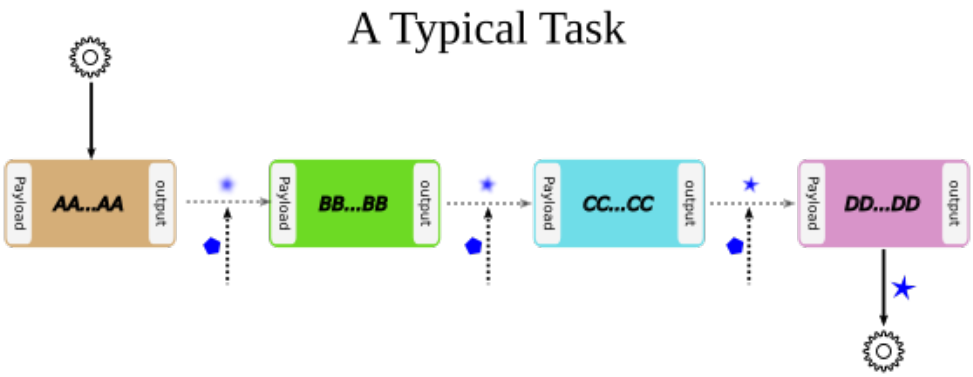
(continued from previous page)

```
      "share": "output/results",
    },
    .
    .
    .
  },
```

The overall message of computes pod is simple yet powerful. Using compute pods, people can autmate things and build on top of others' work.

1.2.3 Tasks

A *task* is a set of independent compute pods loosely chained together to undertake a complex job. The following image demonstrates a visual conception of tasks.



A task is defined in a json file(usually called recipe.json just in the case of compute pods) and has the following look and feel:

```
{
  "name": "some sequence",
  "public": true,
  "pods": [
    "96dd1...59670",
    "e3f8c...55eb4"
  ]
}
```

Where the `pods` property defines a list of compute pods that constitute the task. To run a task you can invoke the Sovr CLI as below:

```
python src/cli.py --recipe foo/recipe.json --run
```

Running a task involves forking and running individual compute pods. After each compute pod is run, the contents of the *output* is copied to the next compute pod's *payload/external* directory, thus enabling dependency of compute pods to each other. To get your feet wet with tasks, there is an example task in `src/templates/tasks/ml/keras/recipe.json` where 5 images are sent to different pre-trained Keras models to be classified.

1.2.4 Quick dive

To make this introduction to compute pods solid, an example is provided here that let's you run and examine a compute pod we have already persisted in Swarm.

1. Set up a user within the FairOS-DFS environment

   We assume that FairOS-DFS tools are located at **./bee/** and our system's architecture is the common 64-bit "x86_64" known as *amd64*

   - Open a terminal window and run(replace the postage block id Swarm gave you with *foobar*)

   ```
   ./bee/dfs-linux-amd64 server --postageBlockId "foobar"
   ```

   - In another terminal tab/window, run

   ```
   ./bee/dfs-cli-linux-amd64
   ```

   Now that you are inside the FairOS-DFS CLI, let's create a user named *sam* or name it as you like

   ```
   user new sam
   ```

   Provide a password for *sam* and exit the FairOS-DFS CLI.

   - Open your favourite text editor and write the following text in it then save it in the Sovr CLI's `src` directory(I hope you've already clonned Sovr CLI, if not please consult *Usage*) as `creds.json`.

   ```
   {
     "username": "sam",
     "password": "sam's password"
   }
   ```

2. Set Golem up as described here *Usage*

3. Fork, run, and persist a compute pod

   While in the same terminal tab/window, make sure you are at Sovr CLI's directory `sovr` and the virtual environment you set up at *Usage* is activated.

   - To fork a compute pod containing a *XCeption* Keras image classification model, run

   ```
   python src/cli.py --fork␣
   ↪a61d11e7335ed41e56494ae4bee5446f7785737938a35454e3190c5ccae283ea
   ```

   Once the forking is complete, you would have `XCeption` directory at your current woking directory, feel free to explore it.

   - To run the forked *XCeption* compute pod, run

   ```
   python src/cli.py --recipe ./XCeption/recipe.json --run
   ```

   This will send your compute pod's stuff to Golem nodes and once done, your compute pod's results are ready at `XCeption/outupt` along with any logs at `XCeption/logs`. For this specific compute pod, the actual result is `XCeption/output/preds.json` which is the top 5 classes the model thought the five input images are.

   - If you are satisfied with the outputs or just interested in saving your compute pod on Swarm, run

   ```
   python src/cli.py --recipe src/XCeption/recipe.json --persist
   ```

If there are no harmless errors, you should get a message on the successful persistence of your compute pod along with a sharing reference key if your recipe's `public` property was *True*.

Congrats, you have completed your very first compute pod journey!

As an alternative to forking, there are some template compute pods and tasks in the `src/templates` directory, feel free to examine them.