daliuge Documentation

Release 3.0.0

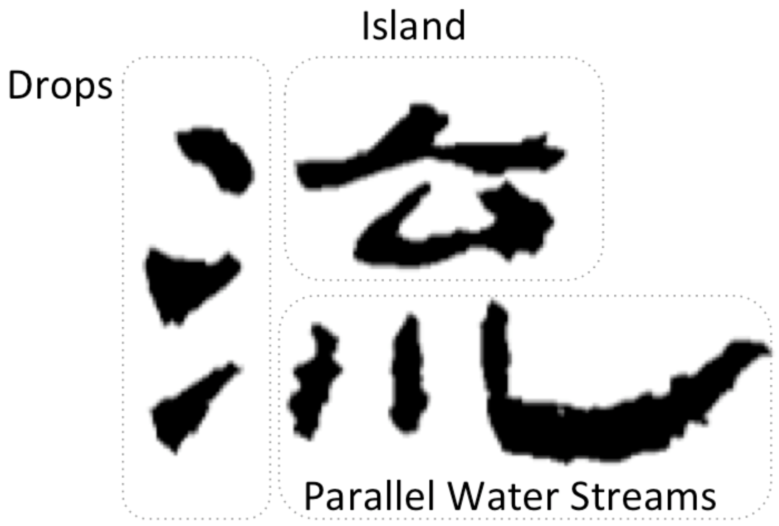
ICRAR

Aug 25, 2023

CONTENTS

Welcome to the Data Activated [1] Graph Engine (DALiuGE).

DALiuGE is a workflow graph execution framework, specifically designed to support very large scale processing graphs for the reduction of interferometric radio astronomy data sets. DALiuGE has already been used for processing large astronomical datasets in existing radio astronomy projects. It originated from a prototyping activity as part of the SDP Consortium called Data Flow Management System (DFMS). DFMS aimed to prototype the execution framework of the proposed SDP architecture. For a complete tour of DALiuGE please read our overview paper. DALiuGE has been used in a project running a full-scale simulation of the Square Kilometre Array dataflow on the ORNL Summit supercomputer.



Parallel *streams* splash onto the (*Data*) *Island*, and turn into *Drops*

Development and maintenance of DALiuGE is currently hosted at ICRAR and is performed by the DIA team.

[1] (pronounced Liu) is the Chinese character for “flow”.

CHAPTER

ONE

INTRODUCTION

The Data Activated (Liu) Graph Engine (DALiuGE) is a workflow graph execution framework, specifically designed to support very large scale processing graphs for the reduction of interferometric radio astronomy data sets. DALiuGE aims to provide a distributed data management platform and a scalable pipeline execution environment to support continuous, soft real-time, data-intensive processing for producing radio astronomy data products.

DALiuGE originated from a prototyping activity as part of the SKA SDP Consortium called Data Flow Management System (DFMS).

The development of DALiuGE is largely based on radio astronomy processing requirements. However, DALiuGE has adopted a generic, data-driven framework architecture potentially applicable to many other data-intensive applications.

DALiuGE stands on shoulders of many previous studies on dataflow, data management, distributed systems (databases), graph theory, and HPC scheduling. DALiuGE has also borrowed useful ideas from existing dataflow-related open sources (mostly *Python*!) such as Luigi, TensorFlow, Airflow, Snakemake, etc. Nevertheless, we believe DALiuGE has some unique features well suited for data-intensive applications:

- Completely *data-activated*, by promoting data *Drops* to become graph "nodes" (no longer just edges) that have persistent states and can consume and raise events
- Integration of data-lifecycle management within the data processing framework
- Separation of concerns between logical graphs (high level workflows) and physical graphs (execution recipes)
- Flexible pipeline component interface, including Docker containers.
- Native multi-core execution out of the box

In *Architecture and Design* we give a glimpse to the main concepts present in DALiuGE. Later sections of the documentation describe more in detail how DALiuGE works. Enjoy!

CHAPTER

TWO

INSTALLATION GUIDE

NOTE: DALiuGE is under heavy development and we are not regularily updating the version on PyPi and DockerHub right now. The currently best way to get going is to install and build from the latest sources which you can get from here:

```
git clone https://github.com/ICRAR/daliuge
cd daliuge
```

2.1 Docker images

The recommended and easiest way to get started is to use the docker container installation procedures provided to build and run the daliuge-engine and the daliuge-translator. We currently build the system in three images:

1. *icrar/daliuge-common* contains all the basic DALiuGE libraries and dependencies.
2. *icrar/daliuge-engine* is built on top of the :base image and includes the installation of the DALiuGE execution engine.
3. *icrar/daliuge-translator* is also built on top of the :base image and includes the installation of the DALiuGE translator.

There are also pre-build images available on dockerHub.

This way we are trying to separate the requirements of the daliuge engine and translator from the rest of the framework, which has a less dynamic development cycle.

The *daliuge-engine* image by default runs a generic daemon, which allows to then start the Master Manager, Node Manager or DataIsland Manager. This approach allows to change the actual manager deployment configuration in a more dynamic way and adjusted to the actual requirements of the environment.

NOTE: This guide is meant for people who are experimenting with the system. It does not cover specific needs of more complex, distributed operational deployments.

2.1.1 Creating the images

Building the three images is easy, just start with the daliuge-common image by running:

```
cd daliuge-common && ./build_common.sh dev && cd ..
```

then build the runtime:

```
cd daliuge-engine&& ./build_engine.sh dev && cd ..
```

and last build the translator:

```
cd daliuge-translator && ./build_translator.sh dev && cd ..
```

2.1.2 Running the images

Running the engine and the translator is equally simple:

```
cd daliuge-engine && ./run_engine.sh dev && cd ..
```

and:

```
cd daliuge-translator && ./run_translator.sh dev && cd ..
```

You can use EAGLE on the URL: https://eagle.icrar.org and point your EAGLE configuration for the translator to http://localhost:8084. Congratulations! You now have access to a complete DALiuGE system on your local computer!

More detailed information about running and controlling the DALiuGE system can be found in the *Startup and Shutdown Guide*.

2.2 Direct Installation

NOTE: For most use cases the docker installation described above is recommended.

2.2.1 Requirements

The DALiuGE framework requires no packages apart from those listed in its

`setup.py`

file, which are automatically retrieved when running it. The spead2 library (one of the DALiuGE optional requirements) however requires a number of libraries installed on the system:

1. boost-python
2. boost-system
3. boost-devel
4. gcc >= 4.8

2.2.2 Installing into host Python

NOTE: DALiuGE requires python 3.7 or later. It is always recommended to install DALiuGE inside a python virtual environment. Make sure that you have on created and enabled. More often than not pip requries an update, else it will always issue a warning. Thus first run:

```
pip install --upgrade pip
```

Like for the docker installation the local installation also follows the same pattern.

Install from GitHub

The following commands are installing the DALiuGE parts directly from github. In this case you won't have access to the sources, but the system will run. First install the daliuge-common part:

```
pip install 'git+https://github.com/ICRAR/daliuge.git#egg&subdirectory=daliuge-common'
```

then install the daliuge-engine:

```
pip install 'git+https://github.com/ICRAR/daliuge.git#egg&subdirectory=daliuge-engine'
```

and finally, if required also install the daliuge-translator:

```
pip install 'git+https://github.com/ICRAR/daliuge.git#egg&subdirectory=daliuge-translator
↪'
```

Install from sources

If you want to have access to the sources you can run the installation in a slightly different way. Again this should be be done from within a virtual environment. First start with cloning the repository:

```
git clone https://github.com/ICRAR/daliuge
```

then install the individual parts:

```
cd daliuge
cd daliuge-common
pip install .
cd ../daliuge-engine
pip install .
cd ../daliuge-translator
pip install .
```

2.3 Notes of the merge project between DALiuGE and Ray

The objective of this activity was to investigate a feasible solution for the flexible and simple deployment of DALiuGE on various platforms. In particular the deployment of DAliuGE on AWS in an autoscaling environment is of interest to us.

Ray (https://docs.ray.io/en/master/) is a pretty complete execution engine all by itself, targeting DL and ML applications and integrating a number of the major ML software packages. What we are in particular interested in is the Ray core software, which states the following:

1. Providing simple primitives for building and running distributed applications.
2. Enabling end users to parallelize single machine code, with little to zero code changes.
3. Including a large ecosystem of applications, libraries, and tools on top of the core Ray to enable complex applications.

Internally Ray is using a number of technologies we are also using or evaluating within DALiuGE and/or the SKA. The way Ray is managing and distributing computing is done very well and essentially covers a number of our target platforms including AWS, SLURM, Kubernetes, Azure and GC.

The idea thus was to use Ray to distribute DALiuGE on those platforms and on AWS to start with, but leave the rest of the two systems essentially independent. In future we may look into a tighter integration between the two.

2.3.1 Setup

Pre-requisites

First you need to install Ray into your local python virtualenv:

```
pip install ray
```

Ray uses a YAML file to configure a deployment and allows to run additional setup commands on both the head and the worker nodes. In general Ray is running inside docker containers on the target hosts and the initial setup thus is to get the Ray docker image from dockerhub. Getting DALiuGE runnning inside that container is pretty straight forward, but requires installation of gcc and that is quite an overhead. Thus we have created a daliuge-ray docker image, which is now available on the icrar dockerhub repo and is donwloaded instead of the standard Ray image.

The rest is then straight forward and just requires to configure a few AWS autoscale specific settings, which includes AWS region, type of head node and type and (maximum and minimum) number of worker nodes as well as whether this is using the Spot market or not. In addition it is required to specify the virtual machine AMI ID, which is a pain to get and different for the various AWS regions.

Starting the DALiuGE Ray cluster

To get DALiuGE up and running in addition to Ray requires just two additional lines for the HEAD and the worker nodes in the YAML file, but there are some caveats as outlined below. With the provided ray configuration YAML file starting a cluster running DALiuGE on AWS is super easy (provided you have your AWS environment set up in place):

```
cd <path_to_daliuge_git_clone>
ray up daliuge-ray.yaml
```

Stopping the cluster is equally simple:

```
ray down daliuge-ray.yaml
```

More for convenience both DALiuGE and Ray require a number of ports to be exposed in order to monitor and connect the various parts. In order to achieve that it is best to attach to a virtual terminal on the Head node and specify all the ports at that point as well:

```
ray attach -p 8265 -p 8001 -p 8000 -p 5555 -p 6666 daliuge-ray.yaml
```

More specifically the command above actually opens a shell inside to the docker container running on the head node AWS instance.

Issues

Bringing the cluster down by default only stops the instances and thus the next startup is quite a bit faster. There is just one 'small' issue: Ray v1.0 has a bug, which prevents the second start to work! That is why the current default setting in daliuge-ray.yaml is to terminate the instances:

```
cache_stopped_nodes: False
```

To stop and start a node manager use the following two commands, replacing the SSH key file with the one created when creating the cluster and the IP address with the public IP address of the AWS node where the NM should be restarted:

```
ssh -tt -o IdentitiesOnly=yes -i /Users/awicenec/.ssh/ray-autoscaler_ap-southeast-2.pem␣
↪ubuntu@54.253.243.145 docker exec -it ray_container dlg nm -s
ssh -tt -o IdentitiesOnly=yes -i /Users/awicenec/.ssh/ray-autoscaler_ap-southeast-2.pem␣
↪ubuntu@54.253.243.145 docker exec -it ray_container dlg nm -v -H 0.0.0.0 -d
```

The commands above also show how to connect to a shell inside the docker container on a worker node. Unfortunately this is not exposed as easily as the connection to the head node in Ray.

Submitting and executing a Graph

This configuration only deploys the DALiuGE engine. EAGLE and a translator need to be deployed somewhere else. When submitting the PG from a translator web interface, the IP address to be entered there is the *public* IP address of the DIM (Ray AWS head instance). After submitting, the DALiuGE monitoring page will pop up and show the progress bar. It is then also possible to click your way through to the sub-graphs running on the worker nodes.

Future thoughts

This implementation is the start of an integration between Ray and DALiuGE. Ray (like the AWS autoscaling) is a *reactive* execution framework and as such it uses the autoscaling feature just in time, when the load exceeds a certain threshold. DALiuGE on the other hand is a *proactive* execution framework and pre-allocates the resources required to execute a whole workflow. Both approaches have pros and cons. In particular in an environment where resources are charged by the second it is desireable to allocate them as dynamically as possible. On the other hand dynamic allocation comes with the overhead of provisioning additional resources during run-time and is thus non-deterministic in terms of completion time. This is even more obvious when using the spot market on AWS. Fully dynamic allocation also does not fit well with bigger workflows, which require lots of resources already at the beginning. The optimal solution very likely is somewhere in the middle between fully dynamic and fully static resource provisioning.

Dynamic workflow allocation

The first step in that direction is to connect the DALiuGE translator with the ray deployment. After the translator has performed the workflow partitioning the resource requirements are fixed and could be used in turn to startup the Ray cluster with the required number of worker nodes. Essentially This would also completely isolate one workflow from another. The next step could be to add workflow fragmentation to the DALiuGE translator and scale the Ray cluster according to the requirements of each of the fragments, rather than the whole workflow. It has to be seen how to trigger the scaling of the Ray cluster just enough ahead of time to be available for the previous workflow fragment to continue without delays.

CHAPTER

THREE

STARTUP AND SHUTDOWN GUIDE

The translator and the engine are separate services and can be installed and run independently.

Depending on how you are intending to run the system startup and shutdown is slightly different.

3.1 For the impatient: Single node DALiuGE

As a developer the following two commands will start both the translator and the engine, including a data island manager (DIM) and a node manager (NM):

```
cd daliuge-translator ; ./run_translator dev ; cd ..
cd daliuge-engine ; ./run_engine dev ; cd ..
```

This is the quickest way to start deploying workflows. Obviously this is limited to a single computer, but certainly useful for testing out the system and developing new components. You can use EAGLE on the URL: https://eagle.icrar.org and point the EAGLE setting (keyboard shortcut 'O') for the translator URL to http://localhost:8084. After submitting your graph to the translator, the translator web interface will be opened in a new tab and you need to adjust the translator settings to point to the URL of the engine you've just started. Since the system is running across docker containers, you need to specify the IP address of the docker host machine, i.e. http://<IP-address>:8001 not localhost. These settings are saved in browser storage, that means that you will have the same settings when coming back. Now you have access to a complete DALiuGE system!

The following paragraphs are providing more detailed guidelines to enable people to start the system on multiple nodes to cover the specific local requirements.

3.2 Starting the docker containers

We are providing convenience scripts to start the docker containers built according to the *Installation Guide*. Depending whether you want to run the development (dev) or the deployment (dep) version of the image there exist different startup options. Starting the translator:

```
cd daliuge-translator
./run_translator.sh dev|dep
```

Similarly starting the engine:

```
cd daliuge-engine
./run_engine.sh dev|dep
```

The main difference between the development and the deployment version is that the development version is automatically strating a data island manager, while the deployment version is not doing that. Both are starting a Node Manager by default (see below). Using the shell scripts is not strictly necessary, but the docker command line is a bit complex.

3.3 Starting and stopping using CLI

If DALiuGE had been installed in a virtual environment of the host system it is possible to start the managers from the command line:

```
dlg dim -H 0.0.0.0 -N localhost -d
```

and a node manager:

```
dlg nm -H 0.0.0.0 -d
```

To stop the managers use:

```
dlg dim -s
```

and:

```
dlg nm -s
```

respectively. The help for the complete CLI is available by just entering dlg at the prompt:

```
 dlg
Usage: /home/awicenec/.pyenv/versions/dlg/bin/dlg [command] [options]

Commands are:
    daemon                   Starts a DALiuGE Daemon process
    dim                      Starts a Drop Island Manager
    fill                     Fill a Logical Graph with parameters
    include_dir              Print the directory where C header files can be found
    lgweb                    A Web server for the Logical Graph Editor
    map                      Maps a Physical Graph Template to resources and produces a␣
↪Physical Graph
    mm                       Starts a Master Manager
    monitor                  A proxy to be used in conjunction with the dlg proxy in␣
↪restricted environments
    nm                       Starts a Node Manager
    partition                Divides a Physical Graph Template into N logical partitions
    proxy                    A reverse proxy to be used in restricted environments to␣
↪contact the Drop Managers
    replay                   Starts a Replay Manager
    submit                   Submits a Physical Graph to a Drop Manager
    unroll                   Unrolls a Logical Graph into a Physical Graph Template
    unroll-and-partition     unroll + partition
    version                  Reports the DALiuGE version and exits

Try $PATH/bin/dlg [command] --help for more details
```

More details about the usage of the CLI can be found in the *CLI User Guide* chapter.

3.3.1 Starting and stopping the managers

DALiuGE is using three different kinds of managers:

1. Node Manager (NM), one per compute node participating in the DALiuGE cluster. The NMs are running all the component wrappers for a single node.
2. Data Island Manager (DIM), which is manageing a (sub-)set of nodes in the cluster. There could be minimum one or maximum as many as NMs Data Island Managers in a deployment. The DIM is also the entity receiving the workflow description from the translator and is then distributing the sections to the NMs.
3. Master Manager (MM), which has the information about all nodes and islands in the deployment. In many deployments the master manager is optional and not really required. If it is necessary, then there is only a single master manager running on the cluster.

Starting a master manager can be done using the dlg command:

```
dlg daemon
```

by default this will also start a NM, but not a DIM.

The managers are spawned off (as processes) from the daemon process, which also exposes a REST interface allowing the user to start and stop managers. The start and stop commands follow the URL pattern[1]:

```
curl -X POST http://localhost:9000/managers/<type>/start
```

and:

```
curl -X POST http://localhost:9000/managers/<type>/stop
```

where <type> is on of [node|dataisland|master]. In case of the DIM (island) it is possible to specify the nodes participating in that specific island. For example:

```
curl -d '{"nodes": ["192.168.1.72","192.168.1.11"]}' -H "Content-Type: application/json"␣
↪-X POST http://localhost:9000/managers/island/start
```

If a manager is already running or already stopped error messages are returned. In order to see which managers are running on a particular node you can use the GET method:

```
curl http://localhost:9000/managers
```

which returns something like:

```
{"master": null, "island": null, "node": 18}
```

In this example there is just a Node Manager running with process ID 18.

[1] The daemon process is listening on port 9000 by default.

3.4 For the independent: Build and run EAGLE

It is also possible to start the EAGLE locally in addition as well. This requires to clone and build the EAGLE repo into a directory separate from the DALiuGE repo:

```
git clone https://github.com/ICRAR/EAGLE
cd EAGLE
./build_eagle dep
```

To start EAGLE:

```
./run_eagle dep
```

This will start the EAGLE docker image built in the previous step and try to open a browser tab.

(NOTE: The usage of the EAGLE visual graph editor is covered in its own documentation).

3.5 Zeroconf

The Master Manager also opens a zeroconf service, which allows the Node Managers to register and deregister and thus the MM is always up to date with the node available in the cluster. NOTE: This mechanism is currently not implemented for the DIMs, i.e. a DIM does not register with the MM automatically. Since it is not possible to guess which NM should belong to which DIM, the NMs also do not register with a DIM. For convenience and as an exception to this rule, when starting the development version of the daliuge-engine image, the single NM is automatically assigned to the DIM on localhost.

CHAPTER

FOUR

BASICS

The DALiuGE system consists of three main components:

1. the EAGLE visual workflow editor,
2. the translator, partitioning and scheduling service (short just *Translator*) and
3. the execution engine (*Engine* for short).

Each of these components can be deployed, run and used independently. The most convenient way of using the system is to drive it from EAGLE. EAGLE is a web application, while the *Translator* service as well as the execution *Engine* services expose RESTful interfaces, which can be used programatically or using command line tools like *curl*. In addition there is a web monitoring tool exposed by the execution engine available as well.

The *Editor* and the *Translator* are fairly lightweight and don't require a lot of resources. In particular the editor can be deployed locally on a user's laptop, but there is also a version running under https://eagle.icrar.org.

The DALiuGE execution engine *can* be run on a laptop as well, but, other than for testing, there is no real good use case to do this. More realistic deployment platforms are large High Performance Computing (HPC) clusters or dynamically scalable environments such as the AWS Elastic Computing Service (ECS) or a Kubernetes cluster. All three components of DALiuGE can be installed and run natively or in docker containers.

4.1 EAGLE

EAGLE is a web-application allowing users to develop complex scientific workflows using a visual paradigm similar to Kepler and Taverna or Simulink and Quartz composer. It also allows groups of scientists to work together on such workflows, version them and prepare them for execution. A workflow in DALiuGE terminology is represented as a graph. In fact the DALiuGE system is dealing with a whole set of different graphs representing the same workflow, depending on the lifecycle state of that workflow. EAGLE just deals with the so-called *Logical Graph* state of a given workflow. EAGLE also offers an interface to the *Translator* and, through the *Translator* also to the *Engine*. For detailed information about EAGLE please refer to the EAGLE basic documentation under https://github.com/ICRAR/EAGLE as well as the detailed usage documentation in readthedocs.

4.2 Translator service

The Translator, partitioning and scheduling service is a FASTApi RESTful service. It takes a *Logical Graph* representation of a workflow and translates that into a *Physical Graph*, which in turn is a directed acyclic graph (DAG). It then uses that DAG and applies some complex heuristic algorithms to distribute the complete DAG on the available platform in an optimized way and also produces an optimzed schedule for that distribution. While the *Logical Graph* might look quite small, it can easily translate into a *Physical Graph* with thousands or millions of nodes and optimizing even just the placement of the nodes of such a system represents a N-P hard problem. The *Physical Graph* can then be send to the *Execution Engine* for execution. If given a translator URL, navigate to URL/docs to find a live interface to test the endpoints manually or read up about their usage.

4.3 Execution Engine

The *Engine* consists of three kinds of RESTful services in order to be able to deal with very large *Physical Graphs* produced by the *Translator*:

1. Master manager
2. Data Island manager
3. Node manager

In addition there is also a small web application, which allows to monitor the progress of DALiuGE execution sessions. The managers are only involved in the deployment of the *Physical Graph*, the execution, once started does not require any central control. During runtime the managers are just monitoring the progress. They also allow to stop or terminate a running workflow. During deployment the master manager uses the partitioning information produced by the *Translator* to split up the *Physical Graph* and send those partitions to node managers. In general each partition will be send to a different compute node, but that is not a rigid mapping.

CHAPTER

FIVE

ARCHITECTURE AND DESIGN

The following sections give an overview of the architecture and design considerations behind DALiuGE.

5.1 Concepts and Background

This section introduces key concepts and motivations underpinning the DALiuGE system.

5.1.1 Dataflow

A traditional dataflow computation model does not explicitly place any control or constraints on the order or timing of operations beyond what is inherent in the data dependencies among compute tasks. The removal of explicit scheduling of compute task in the dataflow model has opened up new (e.g. parallelism) opportunities that are previously masked by "artificial" control flow imposed by applications or programmers. A similar example is the `make` tool, where the programmer focuses on defining each target and its dependencies. The burden of exploring parallelism to efficiently execute many individual compiling tasks in a correct order lies within the responsibility of the `make` utility.

5.1.2 Graph

Following the dataflow model, a computer program can be described by a Directed Graph where the nodes denote compute task, and the edges denote data dependencies between operations. In principle, a dataflow graph consists of edges, nodes (or actors), and tokens. Tokens represent data items and travel across directed edges to be transformed at nodes into other data items (similar to functions). While in theory the dataflow model provides a powerful yet simple formalism to describe parallel computation, early efforts in developing dataflow architecture had to introduce control flow operators (e.g. switch and merge) and data storage mechanism in order to put dataflow models into practice.

5.1.3 Data-activated

In developing DALiuGE, we have extended the "traditional" dataflow model by integrating data lifecycle management, graph execution engine, and cost-optimal resource allocation into a coherent *data-activated* framework. Concretely, we have made the following changes to the existing dataflow model:

- Unlike traditional dataflow models that characterise data as "tokens" moving across directed edges between nodes, we instead model data as the node, elevating them as actors who have autonomy to manage their own lifecycles and trigger appropriate "consumer" applications based on their own internal (persistent) states. In our graph model, both application (task) and data nodes are termed as **Drops**. What are really moving on the edge are *Drop Events*.

- While nodes/actors in the traditional dataflow are stateless functions, we express both computation and data nodes as stateful Drops. Statefulness not only allows us to manage Drops through persistent checkpointing, versioning and recovery after restart, etc., but also enables data sharing amongst multiple processing pipelines in situations like re-processing or commensal observations. All the state information is kept in the Drop wrapper, while the payload of the Drops, i.e. pipeline component algorithms and data, remain stateless.
- We introduced a small number of control flow graph nodes at the logical level such as *Scatter*, *Gather*, *GroupBy*, *Loop*, etc. These additional control nodes allow pipeline developers to systematically express complex data partitioning and event flow patterns based on various requirements and science processing goals. More importantly, we transform these control nodes into ordinary Drops at the physical level. Thus they are nearly transparent to the underlying graph/dataflow execution engine, which focuses solely on exploring parallelisms orthogonal to these control nodes placed by applications. In this way, the data-activated framework enjoys the best from both worlds - expressivity at the application level and flexibility at the dataflow system level.
- Finally, we differentiate between two kinds of dataflow graphs - **Logical Graph** and **Physical Graph**. While the former provides a higher level of abstraction in a resource-independent manner, the latter represents the actual execution plan consisting of inter-connected Drops mapped onto a given set of hardware resources in order to meet performance requirements at minimum cost (e.g. power consumption). In addition we further distinguish between **Logical Graph Templates** and **Logical Graphs** and **Physical Graph Templates** and **Physical Graphs**. The template graph in each of the pairs has a number of free parameters, while in the actual graph everything is fully defined. The free parameters of a **Logical Graph Template** allow changes to the configuration and behaviour of components, but none of those will change the structure and logic of the **Logical Graph**. Similarly free parameters in **Physical Graph Templates** will allow allocation of parts of the graph to certain hardware resources to produce the final **Physical Graph**. The structure of the template is the same as the structure of any **Physical Graph** derived from the same template. Note however that, while changing some of the parameters of a **Logical Graph Template** will not change the structure of the derived **Logical Graph** at all, it can dramatically change the structure of the associated **Physical Graph Template** and **Physical Graph**. For example a scatter construct in a **Logical Graph Template** has exactly the same strcuture for 2 and for 100,000 splits, but the **Physical Graph** will show either 2 or 100,000 branches.

5.2 Drops

Drops are at the center of the DALiuGE. Drops are representations of data and applications, making them manageable by DALiuGE.

5.2.1 Lifecycle

The lifecycle of a Drop is simple and follows the basic principle of writing once, read many times. Additionally, it also allows for data deletion.

A Drop starts in the **INITIALIZED** state, meaning that its data is not present yet. From there it jumps into **COMPLETED** once its data has been written, optionally passing through **WRITING** if the writing occurs *through* DALiuGE (see *Input/Output*). Once in the **COMPLETED** state the data can be read as many times as needed. Eventually, the Drop will transition to **EXPIRED**, denying any further reads. Finally the data is deleted and the Drop moves to the final **DELETED** state. If any I/O error occurs the Drop will be moved to the **ERROR** state.

5.2.2 Events

Changes in a Drop state, and other actions performed on a Drop, will fire named events which are sent to all the interested subscribers. Users can subscribe to particular named events, or to all events.

In particular the *Node Drop Manager* subscribes to all events generated by the Drops it manages. By doing so it can monitor all their activities and perform any appropriate action as required. The Node Drop Manager, or any other entity, can thus become a Graph Event Manager, in the sense that they can subscribe to all events sent by all Drops and make use of them.

5.2.3 Relationships

Drops are connected and create a dependency graph representing an execution plan, where inputs and outputs are connected to applications, establishing the following possible relationships:

1. None or many data Drop(s) can be the *input* of an application Drop; and the application is the *consumer* of the data Drop(s).
2. A data Drop can be a *streaming input* of an application Drop in which case the application is seen as a *streaming consumer* from the data Drop's point of view.
3. None or many Drop(s) can be the *output* of an application Drop, in which case the application is the *producer* of the data Drop(s).
4. An application is never a consumer or producer of another application; conversely a data Drop never produces or consumes another data Drop.

The difference between *normal* inputs/consumers and their *streaming* counterpart is their granularity. In the normal case, inputs only notify their consumers when they have reached the **COMPLETED** state, after which the consumers can open the Drop and read their data. Streaming inputs on the other hand notify consumers each time data is written into them (alongside with the data itself), and thus allow for a continuous operation of applications as data gets written into their inputs. Once all the data has been written, the normal event notifying that the Drop has moved to the **COMPLETED** state is also fired.

5.2.4 Input/Output

I/O can be performed on the data that is represented by a Drop by obtaining a reference to its I/O object and calling the necessary POSIX-like methods. In this case, the data is passing through the Drop instance. The application is free to bypass the Drop interface and perform I/O directly on the data, in which case it uses the data Drop `dataURL` to find out the data location. It is the responsibility of the application to ensure that the I/O is occurring in the correct location and using the expected format for storage or subsequent upstream processing by other application Drops.

DALiuGE provides various commonly used *data components* with their associated I/O storage classes, including in-memory, Apache Arrow Plasma/Flight, file-base, S3 and NGAS storages. It is also possible to access the contant of a plain URL and use that as a data source.

When using and developing a DALiuGE workflow the details of the I/O mechanisms are completely hidden, but users just need to be aware of the differences and limitations of using either of them. Memory and Files or remote data objects are just not really the same in terms of I/O capabilities and performance. The most important difference is between memory and all the other methods, since plain memory really only works for Python and dynamic library based components. A bash component for example simply does not know how to deal with some memory block handed over to it. That is why EAGLE does prevent such connections between components in the first place.

When developing *application* components most of these details are also transparent, as long as the application component is using the provided POSIX-like access mechanisms. It is possible though to bypass those inside a component and perform all I/O independently of the framework. Even on that level there are still two ways, one is to use the provided data url from the framework, but not use the I/O methods. The even more extreme way is to just open some named file or channel without DALiuGE knowing anything about it. This latter way is strongly discouraged, since it will create unpredictable side-effects, which are almost impossible to identify in a large distributed environment. How to use the provided I/O methods from an *application* component is detailed in the *DALiuGE Application Component Developers Guide* chapter.

When developing a new *data* component the developer needs to implement the interface between the DALiuGE POSIX-like methods of the underlying data storage method. This is detailed in the *DALiuGE Data Component Developers Guide* chapter.

5.2.5 Drop Channels

During a DALiuGE workflow execution one application drop produces the data of a data drop, which in turn is consumed by another application drop. That means that data drops are essentially providing the data transfer methods between applications. The DALiuGE translator tries to minimise data movement and thus in many cases no transfer is actually happening, but the data drop transfers to COMPLETED state once it has received all data and passes that event on to the consumer application(s). The consumer applications in turn will use the provided read method to access the data directly.

In cases when data drops are accessed from separate nodes or islands the managers automatically produce a drop proxy on the remote nodes providing a remote method invocation (RMI) interface to allow the producers or consumers to execute the required I/O methods. It's the job of the Master Drop and Island Managers to generate and exchange these proxies and connect them to the correct Drop instances when the graph is deployed to potentially multiple data islands and nodes. If there is no Drop separation within a physical graph partition then its implied that the Drops are going to be executed within a single address space, and, as a result, basic method calls are used between Drop instances.

In addition to the hand-over of the handle to the consumer once the data drop is COMPLETED DALiuGE also supports streaming data directly from one application drop to another during run-time. Like for most streaming applications this is based on the completion of a block of bytes transferred, thus the intermediate data drop still has a meaning and could in priciple be any standard data drop. In practice the only viable solutions are memory based drops, like plain memory, shared memory or Plasma.

5.2.6 Drop Component Interface

The DALiuGE framework uses Docker containers as its primary interface to 3rd party applications. Docker containers have the following benefits over traditional tools management:

1. Portability.
2. Versioning and component reuse.
3. Lightweight footprint.
4. Simple maintenance.

The application programmer can make use of the *DockerApp* which is the interface between a Docker container and the Drop framework. Refer to the documentation for details.

Other applications not based on Docker containers can be written as well. Any application must derive at least from `AppDrop`, but an easier-to-use base class is the `BarrierAppDrop`, which simply requires a `run` method to be written by the developer (see *dlg.rpc* for details). DALiuGE ships with a set of pre-existing applications to perform common operations, like a TCP socket listener and a bash command executor, among others. See *dlg.apps* for more examples. In addition we have developed a stand-alone tool (dlg_paletteGen), which enables the automatic generation of DALiuGE compatible component descriptions from existing code. In this way it is possible to enable to usage of big existing public or propietary libraries of algorithms, like e.g. Astropy within the DALiuGE eco-system.

5.3 Graphs

A processing pipeline or workflow in DALiuGE is described by a Directed Graph where the nodes denote both task (application components) and data (data components). The edges denote execution dependencies between components. Section *Operational concepts* has introduced graph-based functions in DALiuGE. This section provides a more detailed overview of the internals of DALiuGE graphs.

5.3.1 Logical Graph

A *logical graph* is a compact representation of the logical operations and data flow in a processing workflow without being concerned about the underlying hardware resources. Logical graphs are constructed by domain experts who have a clear idea about the steps required to generate the desired science prducts. Many of the components are very domain specific and there are a number of different radio astronomy application and data components available to design logical graphs representing radio astronomy workflows. In addition to simple components DALiuGE also provides a number of complex components to support the encoding of higher level language operations like loop, scatter, gather and group-by. In particular the scatter complex component allows users to encode possible paralellisation of operations and whole sections of the graph. It should be noted though, whether those parts are really executed in parallel or serial depends on the actual deployment and availability of resources capable of the desired parallelism. Such complex components are also referred to as *constructs* in a *logical graph*. *Constructs* are not domain specific, but internally they do refer to simple components, which in turn might be domain specific. For instance a scatter construct might need a very domain specific way of splitting up and preparing the data for every single branch of the scatter.

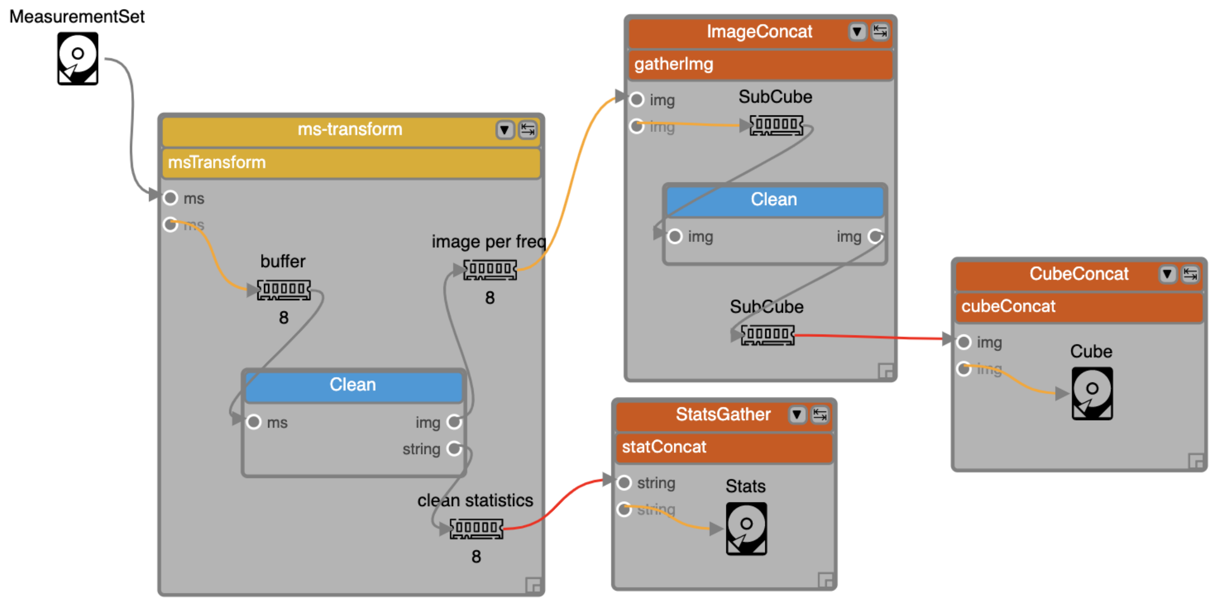


Fig. 5.1: An example of a *logical graph* with various types of data components as well as simple and complex application components. The graph uses two types of data components, *File* and *Memory*, depicted by respective icons. The titles shown with the icons, e.g. MeasurementSet, buffer, SubCube and Stats, refer to the actual content of those data components. There are two simple application components used in this graph, both are refering to the same application called *Clean*. In addition there are four complex components, one scatter construct (ms-transform) and three gather constructs (ImageConcat, CubeConcat and StatsGather). This example can be viewed online in EAGLE. (Note: this requires that you have setup EAGLE with a valid gitHUB access token, see EAGLE help)

Logical Graphs will be translated into *physical graphs* and at that point the component descriptions will be turned into

Drop descriptions (see *Drops*). At execution time these *Drop* descriptions will be instantiated by the execution engine managers.

Component properties

Each component has several associated parameters that users have control over during the development of a *logical graph*. For Application and Data components the **Execution time** and **Data volume** are two important parameters. These properties can be directly obtained from parametric models or estimated from profiling information (e.g. pipeline component workload characterisation) and information about the hardware capabilities.

Complex components (Constructs)

Constructs form the "skeleton" of the *logical graph*, and determine the final structure of the *physical graph* to be generated. DALiuGE currently supports the following flow constructs:

- **Scatter** indicates data parallelism. The group of components inside a *Scatter* construct are consuming a single data partition within the enclosing *Scatter*. The most important user defineable parameter of *Scatter* is `Number of Splits`. In the example in Fig. 5.1, if the `Number of Splits` for `Scatter1` and `Scatter2` are 5 and 4 respectively, the generated *physical graph* will have in total 20 `Data1/Component1/Data3` Drops, but only 5 Drops for the construct `Component 5`, which is inside the `Scatter1` construct but outside `Scatter2`.
- **Gather** indicates data barriers. Constructs inside a *Gather* represent a group of components consuming a sequence of data partitions as a whole. *Gather* has a `Number of Inputs` property, which represents the *Gather* "width", stating how many partitions each *Gather* instance (translated into a `BarrierAppDROP`, see *Drop Component Interface*) can handle. This in turn is used by DALiuGE to determine how many *Gather* instances should be generated in the *physical graph*. *Gather* sometimes can be used in conjunction with *Group By* (see middle-right in Fig. 5.1), in which case, data held in a sequence of groups are processed together by components enclosed by *Gather*. NOTE: The flexibility of *Scatter* and *Gather* constructs allow users to design complex data flow graph patterns by just changing the `Number of Splits` and `Number of Inputs` parameter. However, changing those seemingly simple values may lead to unexpected or even wrong *physical graphs*. Users should thus always verify the _pattern_ of the constructed *physical graphs* on a small but representative scale.
- **Group By** indicates data resorting (e.g. corner turning in radio astronomy). The semantic is analogous to the `GROUP BY` construct used in SQL statement for relational databases, but applied to data Drops. DALiuGE requires that *Group By* is used in conjunction with a nested *Scatter* such that data Drops that are originally sorted in the order of `[outer_partition_id][inner_partition_id]` are resorted as `[inner_partition_id][outer_partition_id]`. In terms of parallelism, *Group By* is comparable to the "static" MapReduce, where the keys used by all Reducers are known a priori. NOTE: As with the *Scatter* and *Gather* constructs, *Group By* constructs provide a very powerful way to change the structure of reduction graphs. Users are advised to always check the resulting *physical graph* _patterns_ for correctness.
- **Loop** indicates iterations. Constructs inside a *Loop* represent a group of components that will be repeatedly executed for a fixed number of times. Although there is also a *Branch* construct, the current DALiuGE implementation does not support dynamic branch conditions inside a *Loop*. Instead, each *Loop* construct has a property named `Number of Iterations` that must be determined at *logical graph* development time, and that determines the number of times the loop is "unrolled". In other words, a `Number of Iterations` number of Drops for each construct inside a *Loop* will be statically generated in the *physical graph*. An example is shown in Fig. 5.2.
- **Branch** indicates conditional execution of sections of a *physical graph*. Branching (as well as loops) are, maybe surprisingly, tricky cases to deal with in a dataflow and DAG environment. Both of them are either explicitly (loop) or potentially (branch) producing cycles and are thus not directly representable as a DAG and thus it is hard to construct a *physical graph*. *Branch* constructs have the additional issue that one side of the branch, depending on the condition, might never be executed. Since the condition result in general is only known at runtime, the *physical graph* that will actually be executed can't be computed upfront and thus scheduling as well as resource

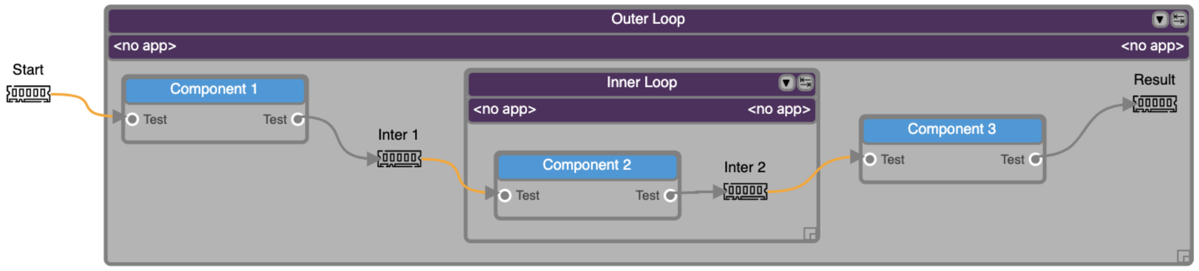


Fig. 5.2: A nested-Loop (outer and inner) example of *logical graph* for a continuous imaging pipeline. This example can be viewed online in DALiuGE.

planning can only be done as an upper (or lower) limit. Although branches do work in DALiuGE, currently in most of the cases the graph execution will not finish, since the engine can't discard whole *physical graph* sections based on a runtime condition and thus the graph will never reach the FINISHED state. We will tackle this issue in a future release.

- **MKN** generalised scatter/gather. While designing the *Scatter*, *Gather* and *Group By* constructs we have found that it is possible to generalise these constructs into what we called *MKN* construct. *MKN* stands for a multiplicity of M externally to the construct, K internal and N on the output side. The MKN constructs are not fully supported throughout the DALiuGE framework yet, but will provide even more powerful ways to construct complex graph patterns. The current implementation is limited to what *Scatter* and *Gather* constructs are doing and thus using those is equivalent and the preferred solution for now.

Repositories

DALiuGE uses EAGLE, a Web-based *logical graph* editor as the default user interface to underlying logical graph and component repositories. Repositories can reside on a local file system, on GitHub or on GitLab. Each *logical graph* is physically stored in those repositories as a JSON-formatted text file. The JSON format is based on a JSON schema and validated against that as well. The JSON file contains the description of the application and data components used in the graph as nodes, a description of the connection between the nodes (edges and connection ports) and also the description of some of the representation properties required to draw the graph.

The repositories also contain so-called *palettes*, which represent a collection of components. Users can pick from those components in EAGLE to draw *logical graph templates*. The differentiation between graphs and palettes is somewhat blurry, since any graph can also be used as a collection of components. However, palettes usually contain a superset of components used in any graph derived from them and thus the distinction is still relevant.

Usage of *Logical Graph Templates* and *Logical Graphs*

EAGLE currently does not explicitely differentiate between a *logical graph* and a *logical graph template*. The only difference between these two are the populated values for some parameters and the relationship between the two is similar to the relationship between classes and instances in an OO language. The graphs in the repositories in general are *logical graph templates* (i.e. classes). The Users can simply load a *logical graph* from one of the repositories and modify the existing parameters before submitting to the translator. In future we will extend the repository functionality of EAGLE to deal with *logical graphs* and *logical graph templates* and also bind the *logical graphs* to execution sessions in the DALiuGE engine.

5.3.2 Translation

While a *logical graph* provides a compact way to express complex processing logic, the complex components or constructs are not directly usable by the underlying graph execution engine and Drop managers. To achieve that, *logical graphs* are translated into *physical graphs*. The translation process makes the parallelism explicit and unrolls loops and creates all Drop descriptions. Drops are essentially instances of the components. It is implemented in the *dlg.dropmake* module.

Basic steps

DropMake in DALiuGE involves the following steps:

- **Validity checking**. Checks whether the *logical graph* is ready to be translated. This step is similar to semantic error checking used in compilers. For example, DALiuGE currently does not allow any cycles in the *logical graph*. Another example is that *Gather* can be placed only after a *Group By* or a *Data* component as shown in Fig. 5.1. Validity errors will be displayed as exceptions on the *logical graph* editor.
- **Construct unrolling**. Unrolls the *logical graph* by (1) creating all necessary Drops (including "artifact" Drops that do not appear in the original *logical graph*), and (2) establishing directed edges amongst all newly generated Drops. This step produces the *physical graph template*.
- **Graph partitioning**. Decomposes the *Physical Graph Template* into a set of logical partitions (a.k.a. *DataIsland*) and generates an order of Drop execution sequence within each partition such that certain performance requirements (e.g. total completion time, total data movement, etc.) are met under given constraints (e.g. resource footprint). An important assumption is that the cost of moving data within the same partition is far less than that between two different partitions. This step produces the *Physical Graph Template Partition*.
- **Resource mapping**. Maps each logical partition onto a given set of resources in certain optimal ways (load balancing, etc.). Concretely, each Drop is assigned a physical resource id (such as IP address, hostname, etc.). This step requires near real-time resource usage information from the computing platform. It also needs Drop managers to coordinate the Drop deployment. In some cases, this mapping step is merged with the previous *Graph partitioning* step to directly map Drops to resources. This step produces the *physical graph*.

DALiuGE supports multiple algorithms implementing the latter two steps and users can choose between them when submitting the *logical graph* to the translator. Under the assumption of uniform resources (e.g. each node has identical capabilities), graph partitioning is equivalent to resource mapping since mapping could simply be implemented as a round-robin allocation to all available resources. For uniform resources, graph partitioning algorithms like e.g. METIS [5] actually support multi-constraints load balancing so that both CPU load and memory usage on each node is roughly similar.

For heterogeneous resources, which DALiuGE does not support yet, usually the graph partitioning is first performed, and then resource mapping refers to the assignment of partitions to different resources based on demands and capabilities using graph / tree-matching algorithms[16] . However, it is also possible that the graph partitioning algorithm directly produces a set of unbalanced partitions "tailored" for those available heterogeneous resources.

In the following context, we use the term **Scheduling** to refer to the combination of both *Graph partitioning* and *Resource mapping*.

Scheduling Algorithms

Optimally scheduling an Acyclic Directed Graph (DAG) that involves graph partitioning and resource mapping as stated in *Basic steps* is known to be an NP-hard problem. DALiuGE has tailored several heuristics-based algorithms from previous research on DAG scheduling and graph partitioning to perform these two steps. These algorithms are currently configured by DALiuGE to utilise uniform hardware resources. Support for heterogenous resources using the list scheduling algorithm will be implemented in a later release. With these algorithms, DALiuGE currently attempts to address the following optimisation goals:

- **Minimise the total cost of data movement** but subject to a given **degree of load balancing**. In this problem, a number N of available resource units (e.g. a number of compute nodes) are given, the translation process aims to produce M DataIslands ($M <= N$) from the *physical graph template* such that (1) the total volume of data traveling between two distinct DataIslands is minimised, and (2) the workload variations measured in aggregated **execution time** (Drop property) between a pair of DataIslands is less than a given percentage p %. To solve this problem, graph partitioning and resource mapping steps are merged into one.
- **Minimise the total completion time** but subject to a given **degree of parallelism** (DoP) (e.g. number of cores per node) that each DataIsland is allowed to take advantage of. In the first version of this problem, no information regarding resources is given. DALiuGE simply strives to come up with the optimal number of DataIslands such that (1) the total completion time of the pipeline (which depends on both execution time and the cost of data movement on the graph critical path) is minimised, and (2) the maximum degree of parallelism within each DataIsland is never greater than the given *DoP*. In the second version of this problem, a number of resources of identical performance capability are also given in addition to the *DoP*. This practical problem is a natural extension of version 1, and is solved in DALiuGE by using the "two-phase" method.
- **Minimise the number of DataIslands** but subject to (1) a given **completion time deadline**, and (2) a given *DoP* (e.g. number of cores per node) that each DataIsland is allowed to take advantage of. In this problem, both completion time and resource footprint become the minimisation goals. The motivation of this problem is clear. In an scenario where two different schedules can complete the processing pipelinewithin, say, 5 minutes, the schedule that consumes less resources is preferred. Since a DataIsland is mapped onto resources, and its capacity is already constrained by a given DoP, the number of DataIslands is proportional to the amount of resources needed. Consequently, schedules that require less number of DataIslands are superior. Inspired by the hardware/software co-design method in embedded systems design, DALiuGE uses a "look-ahead" strategy at each optimisation step to adaptively choose from two conflicting objective functions (deadline or resource) for local optimisation, which is more likely to lead to the global optimum than greedy strategies.

In addition to the automatic deployment and scheduling options, there is also a special construct component available, called 'Exclusive Force Node', to allow users to enforce the placement of certain parts of the graph on a single compute node (NOTE: This is still work-in-progress.). In the case that a scattered section of the graph is enclosed in such an Exclusive Force Node construct, each of the scattered sections will be deployed on a compute node. In case there are not enough compute nodes available to accommodate all the scattered sections, some of them might be deployed (in whole, but together) on a single node. This also shows the risk of using such 'hints': It essentially reduces the degrees of freedom of the scheduling algorithm(s) and thus might turn out to be less optimal at runtime.

5.3.3 Physical Graph

The *Translation* process produces the *physical graph*, which, once deployed and instantiated on the DALiuGE execution engine, becomes a collection of inter-connected Drops in a distributed execution plan across multiple resource units, which we refer to as a *physical graph Instance*. The nodes of a *physical graph Instance* are Drops representing either data or applications, which represent the two base types of Drops. Any two Drops connected by an edge must have different base types, i.e. Drops along a *physical graph Instance* will have alternating base types. This establishes a set of reciprocal relationships between Drops:

1. A data Drop is the *input* of an application Drop; on the other hand the application is a *consumer* of the data Drop.

2. Likewise, a data Drop can be the *output* of an application Drop, in which case the application is the *producer* of the data Drop.
3. Similarly, a data Drop can be a *streaming input* of an application Drop (see *Relationships*) in which case the application is seen as a *streaming consumer* from the data Drop's point of view.

Physical Graphs are the final (and only) graph products that will be submitted to the *Drop Managers*. Once Drop managers accept a *physical graph*, it is their responsibility to instantiate and deploy Drop instances on their managed resources as prescribed in the *physical graph* such as partitioning information (produced during the *Translation*) that allows different managers to distribute graph partitions (i.e. DataIslands) across different nodes by setting up proper *Drop Channels*. Once this instantiation phase is finished, the network of Drops and Drop channels is an exact representation of the *physical graph* and only needs an initial trigger to execute autonomously in a *Execution*. In this sense, the *physical graph Instance* is the actual graph execution engine, the managers are only required to instantiate the *physical graph* and send a trigger event to the start Drop. During execution the managers listen to Drop events and can in turn be used to monitor the execution progress. In order to facilitate the monitoring the Drop Managers also provide web interfaces as well as REST interfaces.

5.3.4 Execution

One of the unique features of DALiuGE is the complete decentralisation of the execution. A *Physical Graph Instance* has the ability to advance its own execution. This is internally implemented via the Drop event mechanism as follows:

1. Once a data Drop moves to the COMPLETED state it will fire an event to all its consumers. Consumers (applications) will then assert if they can start their execution depending on their nature and configuration. A specific type of application is the `BarrierAppDROP`, which waits until all its inputs are in the **COMPLETED** state to start its execution.
2. On the other hand, data Drops receive an event every time their producers finish their execution. Once all the producers of a Drop have finished, the Drop moves itself to the **COMPLETED** state, notifying its consumers, and so on.

Failures on applications and data Drops are transmitted likewise automatically via events. Data Drops move to **ERROR** if any of its producers move to **ERROR**, and application Drops move the **ERROR** if a given input error threshold (defaults to 0) is passed (i.e., when more than a given percentage of inputs move to **ERROR**) or if their execution fails. This way whole branches of execution might fail, but after reaching a gathering point the execution might still resume if enough inputs are present.

5.3.5 Parallelism

Speaking about execution, DALiuGE also exhibits multiprocessing of drops using Python's native multiprocessing library. If enabled, drops are launched for execution on their own threads and all memory-drops become shared-memory-drops which write to `/dev/shm`. While relatively robust, one should be careful to ensure safe-access to memory-drops in this case, opting to use scatter/gather or other explicit aggregation stages where necessary.

5.3.6 Shared Memory

In order to enable truly parallel Python components, a lightweight method to share data between system processes is needed. This approach (with caveats) essentially defeats the GIL and therefore requires an explanation; but first, the caveats.

- SharedMemoryDROPs are not thread-safe - simultaneous access (writing or reading) incurrs undefined behaviour - use other, more heavy-weight data stores if necessary.
- You must be using Python 3.8 or newer - our implementation relies on features only included from 3.8 onwards.

- Windows is not supported - but if enough demand was present, it could be implemented back in.

Onto the solution. To share memory between processes, we create files in `/dev/shmem` for each drop, brokered by an imaginatively named `SharedMemoryManager`. Each DALiuGE Node Manager has an associated SharedMemoryManager which addresses shared memory by `session/uid` pairs. The need to create *named* blocks of shared memory necessitates the development of our own manager, rather than using the standard implementation. Upon session completion (or failure), the SharedMemoryManager destroys all shared memory blocks associated with that session. SharedMemoryDROPs can grow or shrink automatically and arbitrarily or be provided a specific size to use. Their default size is 4096 bytes. Shrunk memory will be truncated, grown blocks will contain a copy of the old data.

As mentioned previously, if DALiuGE is configured to utilise multiple cores, there is no need to specifically use SharedMemoryDROPs, InMemoryDROPs will be switched automatically. However, if the need arises, one can specifically use SharedMemoryDROPs.

5.3.7 Environment Variables

Often, several workflow components rely on shared global configuration values, usually stored in imaginatively named configuration files. DALiuGE supports this approach, of course, but offers additional, more transparent options. The EnvironmentVarDROP is a simple key-value store accessible at runtime by all drops in a workflow. One can include multiple `EnivronmentVarDROP``s in a single workflow, **but each variable store must have a unique name**. In a logical graph, reference environment variables as component or application parameters with the following syntax:  ``${EnvironmentVarDROP_Name}.{Variable_name}` The translator and engine handle parsing and filling of these parameters automatically. Variables beginning with `$DLG_`, such as `$DLG_ROOT` are an exception which are handled seperately. These variables come from the deployment themselves and are fetched from the deployment environment at runtime.

One may also access these variables individually at runtime using the `get_environment_variable(key)` function, which accepts a key in the syntax mentioned above, returning `None` if the variable store or key does not exist.

5.4 Drop Managers

The runtime environment of DALiuGE consists on a hierarchy of *Drop Managers*. Drop Managers offer a standard interface to external entities to interact with the runtime system, allowing users to submit physical graphs, deploy them, let them run and query their status.

Drop Managers are organized hierarchically, mirroring the topology of the environment hosting them, and thus enabling scalable solutions. The current design is flexible enough to add more intermediate levels if necessary in the future. The hierarchy levels currently present are:

1. A *Node Drop Manager* is started on every compute node in the cluster.
2. Compute nodes are grouped into *Data Islands*, and thus a *Data Island Drop Manager* exists at the Data Island level.
3. On top of the Data Islands a *Master Drop Manager* can be deployed.

5.4.1 Sessions

The Drop Managers' work is to manage and execute physical graphs. Because more than one physical graph can potentially be deployed in the system, Drop Managers introduce the concept of a *Session*. Sessions represent a physical graph execution, which are completely isolated from one another. This has two main consequences:

1. Submitting the same physical graph to a Drop Manager will create two different sessions
2. Two physical graph executions can run at the same time in a given Drop Manager.

Sessions have a simple lifecycle: they are first created, then a physical graph is attached into them (optionally by parts, or all in one go), after which the graph can be deployed (i.e., the Drops are created). This leaves the session in a running state until the graph has finished its execution, at which point the session is finished and can be deleted.

5.4.2 Node Drop Manager

Node Drop Managers sit at the bottom of the Drop management hierarchy. They are the direct responsible for creating and deleting Drops, and for ultimately running the system.

The Node Drop Manager works mainly as a collection of sessions that are created, populated and run. Whenever a graph is received, it checks that it's valid before accepting it, but delays the creation of the Drops until deployment time. Once the Drops are created, the Node Drop Manager exposes them via a proxy to allow remote method executions on them.

The node manager is also responsible for launching drops on separate processes and managing shared memory access between them.

5.4.3 Data Island Drop Manager

Data Island Drop Managers sit on top of the Node Drop Managers. They follow the assumed topology where a set of nodes is grouped into a logical *Data Island*. The Data Island Drop Manager is the public interface of the whole Data Island to external users, relaying messages to the individual Node Drop Managers as needed.

When receiving a physical graph, the Data Island Drop Manager will first check that the nodes of the graph contain all the necessary information to route them to the correct Node Drop Managers. At deployment time it will also make sure that the inter-node Drop relationships (which are invisible from the Node Drop Managers' point of view) are satisfied by obtaining Drop proxies (using remote procedure calls) and linking them correspondingly.

5.4.4 Master Drop Manager

The Master Drop Manager works exactly like the Data Island Drop Manager but one level above. At this level a set of Data Islands are gathered together to form a single group of which the Master Drop Manager is the public interface.

5.4.5 Interface

All managers in the hierarchy expose a REST interface to external users. The interface is exactly the same independent of the level of the manager in the hierarchy.

The hierarchy contains the following entry points:

```
GET    /api
POST   /api/sessions
GET    /api/sessions
GET    /api/sessions/<sessionId>
```

(continues on next page)

(continued from previous page)

```
DELETE /api/sessions/<sessionId>
GET    /api/sessions/<sessionId>/status
POST   /api/sessions/<sessionId>/deploy
GET    /api/sessions/<sessionId>/graph
GET    /api/sessions/<sessionId>/graph/status
POST   /api/sessions/<sessionId>/graph/append
```

The interface indicate the object with which one is currently interacting, which should be self-explanatory. `GET` methods are queries performed on the corresponding object. `POST` methods send data to a manager to create new objects or to perform an action. `DELETE` methods delete objects from the manager.

Of particular attention is the `POST /api/sessions/<sessionId>/graph/append` method used to feed a manager with a physical graph. The content of such request is a JSON list of objects, where each object contains a full description of a Drop to be created by the manager.

5.4.6 Clients

Python clients are available to ease the communication with the different managers. Apart from that, any third-party tool that talks the HTTP protocol can easily interact with any of the managers.

5.5 Data Lifecycle Manager

As mentioned in *Introduction* and *Data-activated* DALiuGE also integrates a data lifecycle management within the data processing framework. Its purpose is to make sure the data is dealt with correctly in terms of storage, taking into account how and when it is used. This includes, for instance, placing medium- and long-term persistent data into the optimal storage media, and to remove data that is not used anymore.

The current DALiuGE implementation contains a Data Lifecycle Manager (DLM). Because of the high coupling that is needed with all the Drops the DLM is contained within the *Node Drop Manager* processes, and thus shares the same memory space with the Drops it manages. By subscribing to events sent by individual Drops it can track their state and react accordingly.

The DLM functionalities currently implemented in DALiuGE are:

1. Automatically expire Drops; i.e., moves them from the **COMPLETED** state into the **EXPIRED** state, after which they are not readable anymore.
2. Automatically delete data from Drops in the **EXPIRED** state, and move the Drops into the **DELETED** state.
3. Persist Drops' states in a registry (currently implemented with an in-memory registry and a RDBMS-based registry).

How and when a Drop is expired can be configured via two per-Drop, mutually exclusive methods:

1. A `lifetime` can be set in a Drop indicating how long should it live, and after which it should be moved to the **EXPIRED** state, regardless of whether it is still being used or not.
2. A `expire_after_use` flag can be set in a Drop indicating that it should be expired right after all its consumers have finished executing.

5.6 Scientific Reproducibility

Under construction

The scientific reproducibility of computational workflows is a fundamental concern when conducting scientific investigations. Here, we outline our approach to increasing scientific confidence in DALiuGE workflows. Modern methods create a deterministic computing environment through careful software versioning and containerization. We suggest testing equivalence between carefully selected provenance information to complement such approaches.

Doing so allows any workflow system which generates identical provenance information can claim to re-create some aspect of the original workflow execution. Drops provide component-specific provenance information at runtime and throughout graph translation.

Additionally, a novel hash-graph (BlockDAG) method captures the relationships between components by linking provenance throughout an entire workflow. The resulting signature completely characterizes a workflow allowing for constant time provenance comparison.

We refer a motivated reader to the related thesis.

5.6.1 R-Mode Standards

Each drop's provenance information defines what a workflow signature claims. Inspired and extending current workflow literature, we define seven R-modes. R-mode selection occurs when submitting a workflow to DALiuGE for initial filling and unrolling; DALiuGE handles everything else automatically.

Additionally, the ALL mode will generate a signature structure containing separate hash graphs for all supported modes, which is a good choice when experimenting with new workflow concepts or certifying a particular workflow version.

Conversely, the NOTHING option avoids all provenance collection and processing, which may be of performance interest. For now, this is also the default option if no rmode is specified.

Rerunning

A workflow reruns another if they execute the same logical workflow; their logical components and dependencies match. At this standard, the runtime information is simply an execution status flag; the translate-time information is logical template data excluding physical drops structurally.

When scaling up an in-development workflow or deploying to a new facility asserting that executions rerun the original workflow build confidence in the workflow tools. Rerunning is also useful where data scale and contents change, like an ingest pipeline.

Repeating

A workflow repeats another if they execute the same logical workflow and a principally identical physical workflow; their logical components, dependencies, and physical tasks match. At this standard, the runtime information is still only an execution flag, and translate-time information includes component parameters (in addition to rerunning information) and includes all physical drops structurally.

Workflows with stochastic results need statistical power to make scientific claims. Asserting workflow repetitions allows using results in concert.

Recomputing

A workflow recomputes another if they execute the same physical workflow; their physical tasks and dependencies match precisely. In addition to repetition information, a maximal amount of detail for computing drops is stored at this standard.

Recomputation is a meticulous approach that is helpful when debugging workflow deployments.

Reproducing

A workflow reproduces another if their scientific information match. In other words, the terminal data drops of two workflows match in content. The precise mechanism of establishing comparable data need not be a naive copy but is a domain-specific decision. At this standard, runtime and translate-time data only include data-drops structurally. At runtime, data drops are expected to provide a characteristic summary of their contents

Reproductions are practical in asserting whether a given result can be independently reported or to test an alternate methodology. An alternate methodology could mean an incremental change to a single component (somewhat akin to regression testing) or testing a vastly different workflow approach.

Replicating - Scientifically

A scientific replica reruns and reproduces a workflow execution.

Scientific replicas establish a workflow design as a gold standard for a given set of results.

Replicating - Computationally

A computational replica recomputes and reproduces a workflow execution.

Computational replicas are useful if performing science on workflows directly (performance claims etc.)

Replicating - Totally

A total replica repeats and reproduces a workflow execution.

Total replicas allow for independent verification of results, adding direct credibility to results coming from a workflow. Moreover, if a workflow's original deployment environment is unavailable, a total replica is the most robust assertion possibly placed on a workflow.

5.6.2 Technical approach

The fundamental primitive powering workflow signatures are Merkle trees and Block directed acyclic graphs (BlockDAGs). These data structures cryptographically compress provenance and structural information. We describe the primitives of our approach and then their combination. The most relevant code directory is found under `dlg.common.reproducibility`.

Provenance data is stored internally within the graph data-structure throughout translation and execution.

In the logical graph structure (dictionary) this information is keyed under 'reprodata'. In the physical graph (template) structure this information is appended to the end of the droplist.

Following graph execution, the reprodata is written to a log file, alongside the associated execution logs ($DLG_ROOT/logs).

If the specified rmode is 'NOTHING', no reprodata is appended at any stage in translation and execution.

Merkle Trees

A Merkle tree is essentially a binary tree with additional behaviours. Leaves store singular data elements and are hashed in pairs to produce internal nodes containing a signature. These internal nodes are recursively hashed in pairs, eventually leaving a single root node with a signature for its entire sub-tree.

Merkle tree comparisons can find differing nodes in a logarithmic number of comparisons and find their use in version control, distributed databases and blockchains.

We store information for each workflow component in a Merkle tree.

BlockDAGs

BlockDAGs are our term for a hash graph. Each node takes the signature of a previous block(s) in addition to new information, hashes them all together to generate a signature for the current node. We overlay BlockDAGs onto DALiuGE workflow graphs; the edges between components remain, descendant components receive their parents' signatures to generate their signatures, which are passed on to their children.

The root of a Merkle tree formed by the signatures of workflow leaves acts as the full workflow signature.

One could, in principle, do away with these cryptographic structures, but utilizing Merkle trees and BlockDAGs make the comparison between workflow executions constant time independent of workflow scale or composition.

Runtime Provenance

Each drop implements a series of `generate_x_data`, where `x` is the name of a particular standard (defined below). At runtime, drops package up pertinent data then sent to its manager, percolating up to the master responsible for the drop's session, which then packages the final BlockDAG for that workflow execution. The resulting signature structure is written to a file stored alongside that session's log file.

In general, specialized processing drops need to implement a customized `generate_recompute_data` function, and data drops need to implement a `generate_reproduce_data` function.

Translate-time Provenance

DALiuGE can generate BlockDAGs and an associated signature for a workflow at each stage of translations from logical to physical layers. Passing an `rmode` flag (defined below) to the `fill` operation, from that point forward, DALiuGE will capture provenance and pertinent information automatically, storing this information alongside the graph structure itself.

The *pertinent* information is defined in the `dlg.common.reproducibility.reproducibility_fields` file, which will need modification whenever an entirely new type of drop is added (a relatively infrequent occurrence).

Signature Building

The algorithm used to build the blockDAG is a variant of Kahn's algorithm for topological sorting. Nodes without predecessors are processed first, followed by their children, and so on, moving through the graph.

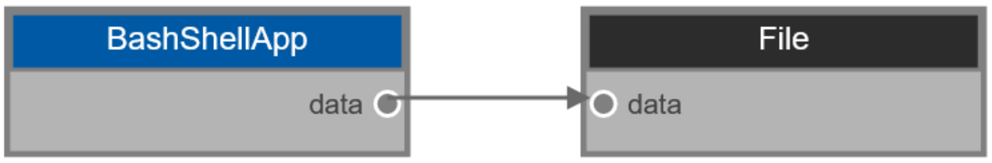
This operation takes time linear in the number of nodes and edges present in the graph at all layers. Building the MerkleTree for each drop is a potentially expensive operation, dependent on the volume of data present in the tree. This is a per-drop consideration, and thus when implementing `generate_reproduce_data`, be wary of producing large data volumes.

5.6.3 Hello World Example

We present a simple example based on several 'Hello world' workflows. First, we present the workflows and signatures for all rmodes and discuss how they compare.

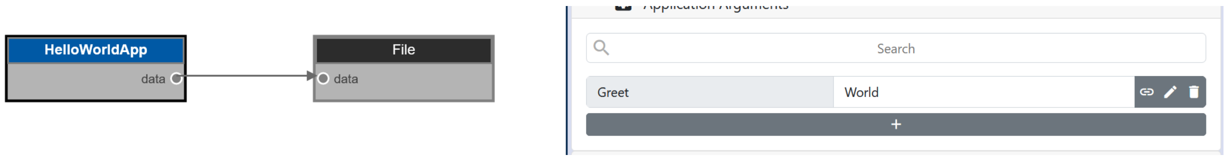
Hello World Bash

This workflow is comprised of a bash script, writing text to a file - Specifically *echo 'Hello World' > %o0*



Hello World Python

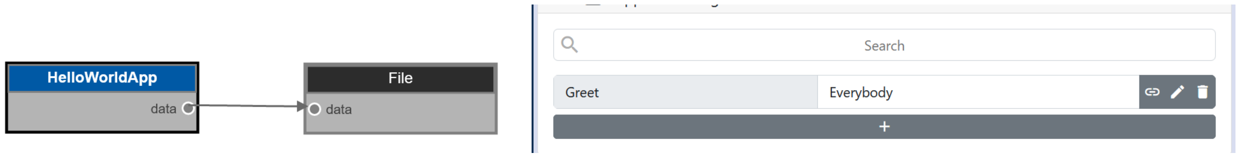
This workflow is comprised of a single python script and a file. This function writes 'Hello World' to the linked file.



Greet	World

Hello Everybody Python

This workflow is again comprised of a single python script and file. This function writes 'Hello Everybody' to the linked file.



Greet	Everybody

Signature Comparisons

By comparing the hashes of each workflow together, we arrive at the following conclusions:

Table 5.1: Workflow Hashes

Workflow	Rerun	Repeat	Recompute	Reproduce	Replicate-SCI	Replicate-COMP	Replicate-TOTAL
HelloWorldBash	d35a6ee278da[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
HelloWorldPython	6413ca52dc80[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
HelloEverybodyPython	6413ca52dc80[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

- HelloEverybodyPython and HelloWorldPython are Reruns
- No two workflows are repetitions
- No two workflows are recomputations
- HelloWorldBash and HelloWorldPython reproduce the same results
- No two workflows are replicas.

Testing for repetitions is primarily useful when examining stochastic workflows to take their results in concert with confidence. Testing or replicas is useful when moving between deployment environments or verifying the validity of a workflow. When debugging a workflow or asserting if the computing environment has changed, recomputations and computational replicas are of particular use.

This simple example scratches the surface of what is possible with a robust workflow signature scheme.

5.6.4 Graph Certification

'Certifying' a graph involves generating and publishing reproducibility signatures. These signatures can be integrated into a CI/CD pipeline, used during executions for verification or during late-stage development when fine-tuning graphs.

By producing and sharing these signatures, subsequent changes to execution environment, processing components, overall graph design and data artefacts can be easily and efficiently tested.

Certifying a Graph

The process of generating and storing workflow signatures is relatively straightforward.

- From the root of the graph-storing directory (usually a repository) create a `/reprodata/[GRAPH_NAME]` directory.
- Run the graph with the `ALL` reproducibility flag, and move the produced reprodata.out file to the previously created directory.
- (optional) Run from `dlg.common.reproducibility.reprodata_compare.py` script with this file as input to generate a summary-csv file

In subsequent executions or during CI/CD scripts: * Note the reprodata.out file generated during the test execution * Run `dlg.common.reproduciblity.reprodata_compare.py` with the published `reprodata/[GRAPH_NAME]` directory and newly generated signature file * The resulting `[SESSION_NAME]-comparison.csv` will contain a simple True/False summary for each RMode, for use at your discretion.

What is to be expected?

In general, all but `Recomputation` and `Replicate_Computational` rmodes should match, moreover:

- A failed `Rerun` indicates some fundamental structure is different
- A failed `Repeat` indicates changes to component parameters or a different execution scale
- A failed `Recomputation`~ indicates some runtime environment changes have been made
- A failed `Reproduction` indicates data artefacts have changed
- A failed `Scientific Replication` indicates a change in data artefacts or fundamental structure
- A failed `Computational Replication` indicates a change in data artefacts or runtime environment
- A failed `Total Replica` indicates a change in data artefacts, component parameters or different execution scale

When attempting to re-create some known graph-derived result, `Replication` is the goal. In an operational context, where data changes constantly, `Reruning` is the goal When conducting science across multiple trials, `Repeating` is necessary to use the derived data arte-facts in concert.

Tips on Making Graphs Robust

The most common 'brittle' aspect of graphs are hard-coded paths to data resources and access to referenced data. This can be ameliorated by:

- Using the `$DLG_ROOT` keyword in component parameters as a base path.
- Providing comments on where to find referenced data artefacts
- Providing instructions on how to build referenced runtime libraries (in the case of Dynlib drops).

5.6.5 Creating New Drop Types

Drops must supply provenance data on demand as part of our scientific reproducibility efforts. When implementing entirely new drop types, ensuring the availability of appropriate information is essential to continue the feature's power.

Drops supply provenance information for various 'R-modes' through `generate_x_data` methods. In the case of application drops specifically, the `generate_recompute_data` method may need overriding if there is any specific information for the exact replication of this component. For example, Python drops may supply their code or an execution trace.

In the case of data drops, the `genreate_reproduce_data` may need overriding and should return a summary of the contained data. For example, the hash of a file, a list of database queries or whatever information deemed characteristic of a data-artefact (perhaps statistical information for science products).

Additionally, if adding an entirely new drop type, you will need to create a new drop category in `dlg.common.__init__.py` and related entries in `dlg.common.reproducibility.reproducibility_fields.py`.

5.7 References

1. Nikhil, R.S., 1990. Executing a program on the MIT tagged-token dataflow architecture. Computers, IEEE Transactions on, 39(3), pp.300-318.
2. Iverson, M.A., Özgüner, F. and Follen, G.J., 1996, August. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In High Performance Distributed Computing, 1996., Proceedings of 5th IEEE International Symposium on (pp. 263-270). IEEE.
3. Gaussier, E., Glesser, D., Reis, V. and Trystram, D., 2015, November. Improving backfilling by using machine learning to predict running times. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (p. 64). ACM.
4. Chaudhary, V. and Aggarwal, J.K., 1993. A generalized scheme for mapping parallel algorithms. Parallel and Distributed Systems, IEEE Transactions on, 4(3), pp.328-346.
5. Karypis, G. and Kumar, V., 1998. Multilevelk-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed computing, 48(1), pp.96-129.
6. Topcuoglu, H., Hariri, S. and Wu, M.Y., 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. Parallel and Distributed Systems, IEEE Transactions on, 13(3), pp.260-274.
7. Kwok, Y.K. and Ahmad, I., 1999. Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys (CSUR), 31(4), pp.406-471.

8. Yang, T. and Gerasoulis, A., 1994. DSC: Scheduling parallel tasks on an unbounded number of processors. Parallel and Distributed Systems, IEEE Transactions on, 5(9), pp.951-967.
9. Sarkar, V., 1989. Partitioning and Scheduling Parallel Programs for Multiprocessors. MIT Press
10. https://en.wikipedia.org/wiki/Antichain
11. Mohan, C., Pirahesh, H., Tang, W.G. and Wang, Y., 1994. Parallelism in relational database management systems. IBM Systems Journal, 33(2), pp.349-371.
12. Wang, Y., 1995, September. DB2 query parallelism: Staging and implementation. In Proceedings of the 21th International Conference on Very Large Data Bases (pp. 686-691). Morgan Kaufmann Publishers Inc.
13. Mehta, M. and DeWitt, D.J., 1995, September. Managing intra-operator parallelism in parallel database systems. In VLDB (Vol. 95, pp. 382-394).
14. Kalavade, A. and Lee, E.A., 1994, September. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In Proceedings of the 3rd international workshop on Hardware/software co-design (pp. 42-48). IEEE Computer Society Press.
15. Liou, J.C. and Palis, M.A., 1997, April. A comparison of general approaches to multiprocessor scheduling. In Parallel Processing Symposium, 1997. Proceedings., 11th International (pp. 152-156). IEEE.
16. Jeannot, E., Mercier, G. and Tessier, F., 2014. Process placement in multicore clusters: Algorithmic issues and practical techniques. Parallel and Distributed Systems, IEEE Transactions on, 25(4), pp.993-1002.
17. Bokhari, S.H., 2012. Assignment problems in parallel and distributed computing (Vol. 32). Springer Science & Business Media
18. R. Wang, et al., "Processing Full-Scale Square Kilometre Array Data on the Summit Supercomputer," in 2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Atlanta, GA, US, 2020 pp. 11-22. doi: 10.1109/SC41405.2020.00006

CHAPTER

SIX

OPERATIONAL CONCEPTS

As mentioned above, DALiuGE has been developed to enable processing of data from the future Square Kilometre Array (SKA) observatory. To support the SKA operational environment DALiuGE provides eight Graph-based functions as shown in Fig. 6.1. The implementation of these operational concepts in general does not restrict the usage of DALiuGE for other use cases, but it is still taylored to meet the SKA requirements.

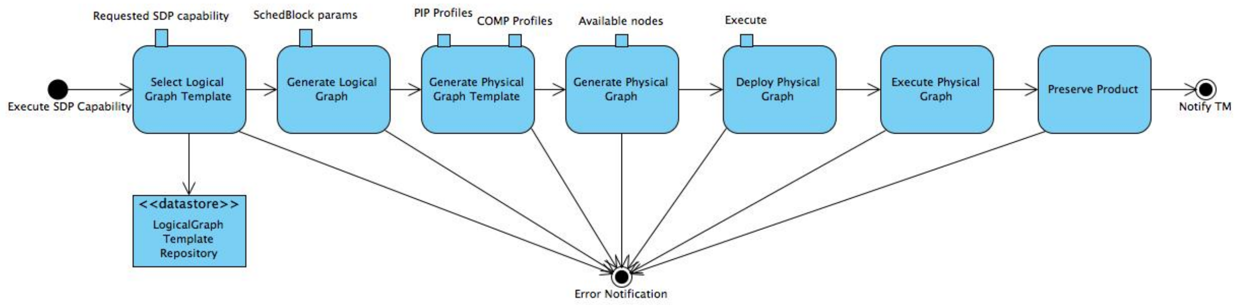


Fig. 6.1: Graph-based Functions of the DALiuGE Prototype

The *Graphs* section describes the implementation details for each function. Here we briefly discuss how they work together to fullfill the SKA requirements.

- First of all, the *Logical Graph Template* (topleft in Fig. 6.1) represents high-level data processing capabilities. In the case of the SKA Data Processor, they could be, for example, “Process Visibility Data” or “Stage Data Products”.
- Logical Graph Templates are managed by *LogicalGraph Template Repositories* (bottomleft in Fig. 6.1). The logical graph template is first selected from this repository for a specific pipeline and is then populated with parameters derived from the detailed description of the scheduled science observation. This generates a *Logical Graph*, expressing a workflow with resource-oblivious dataflow constructs.
- Using profiling information of pipeline components executed on specific hardware resources, DALiuGE then “translates” a Logical Graph into a *Physical Graph Template*, which prescribes a manifest of all Drops without specifying their physical locations.
- Once the information on resource availability (e.g. compute node, storage, etc.) is presented, DALiuGE associates each Drop in the physical graph template with an available resource unit in order to meet pre-defined requirements such as performance, cost, etc. Doing so essentially transforms the physical graph template into a *Physical Graph*, consisting of inter-connected Drops mapped onto a given set of resources.
- All four graph varieties are serializable as JSON strings, that is also how graphs are stored in repositories and transferred.
- Before an observation starts, the DALiuGE engine de-serializes a physical graph JSON string and turns all the nodes into Drop objects and then deploys all the Drops onto the allocated resources as per the location information stated in the physical graph. The deployment process is facilitated through *Drop Managers*, which are daemon

processes managing the deployment of Drops onto the designated resources. Note that the *Drop Managers* do _not_ control the Drops or the execution, but they do monitor the state of them during the execution.

- Once an observation starts, the graph *Execution* cascades down the graph edges through either data Drops that triggers its next consumers or application Drops that produces its next outputs. When all Drops are in the **COMPLETED** state, some data Drops are persistently preserved as Science Products by using an explicit persist consumer, which very likely will be specifically dedicated to a certain science data product.

6.1 Deployment Scenarios

The three components described in the *Basics* section allow for a very flexible deployment. In a real world deployment there will always be one data island manager, zero or one master managers, and as many node managers as there are computing nodes available to the DALiuGE execution engine. In very small deployments one node manager can take over the role of the master manager as well. For extremely large deployments DALiuGE supports a hierarchy of island managers to be deployed, although even with 10s of millions of tasks we have not seen the actual need to do this. Note that the managers are only deploying the graph, the execution is completely asynchronous and does not require any of the higher level managers to run. Even the *manager functionality* of the node manager is not required at run-time.

The primary usage scenario for the DALiuGE execution engine is to run it on a large cluster of machines with very large workflows of thousands to millions of individual tasks. However, for testing and small scale applications it is also possible to deploy the whole system on a single laptop or on a small cluster. It is also possible to deploy the whole system or parts of it on AWS or a Kubernetes cluster. For instance EAGLE and/or the *translator* could run locally, or on a single AWS instance and submit the physical graph to a master manager on some HPC system. This flexible deployment is also the reason why the individual components are kept well separated.

The translator is able to determine which of the following options is available given the selected deployment endpoint.

6.1.1 Deployment in HPC Centers

When trying to deploy DALiuGE inside a HPC centre the basic concept as described above does not apply, since in general it is not possible to have the managers running on nodes in a daemon-like way. Typically a user has to submit a job into a batch queue system like SLURM or Torque and that is pretty much all that can be done by a normal user. In order to address this use case, the DALiuGE code base contains example code (daliuge-engine/dlg/deploy/pawsey/start_dfms_cluster.py) which essentially allows to submit not just the workflow, but also the DALiuGE engine as a job. The first thing that job is then doing is to start the managers and then submit the graph. It also allows to start a proxy server, which provides access to the managers' web interfaces via an external machine in order to be able to monitor the running graph. The best way to get access to the DALiuGE code base is to ask the support team to create a load module specifically for DALiuGE. If that is not possible, then users can just load an appropriate Python version (3.7 or 3.8) and install DALiuGE locally. In many cases it is not possible to run docker containers on HPC infrastructure.

6.1.2 Deployment with OpenOnDemand

OpenOnDemand (OOD) is a system providing an interactive interface to remote compute resources. It is becoming increasingly popular with a number of HPC centers around the world. The two Australian research HPC centers Pawsey and NCI are planning to roll it out for their users. Independently we had realized that DALiuGE is missing a authentication, authorization and session management system and started looking into OOD as a solution for this. After a short evaluation we have started integrating OOD into the deployment for our small in-house compute cluster. In order to make this work we needed to implement an additional interface between the translator running on an external server (e.g. AWS) and OOD and then further on into the (SLURM) batch job system. This interface code is currently in a separate private git repository, but will be released as soon as we have finished testing it. The code mimics the DALiuGE data island manager's REST interface, but instead of launching the workflow directly it prepares a SLURM job submission script and places it into the queue. Users can then use the standard OOD web-pages to monitor the jobs and get access to the logs and results of the workflow execution. OOD allows the integration of multiple compute resources, including Kubernetes and also (to a certain degree) GCP, AWS and Azure. Once configured, users can choose to submit their jobs to any of those. Our OOD interface code has been implemented as an OOD embedded Phusion Passenger Flask application, which is WSGI compliant. Very little inside that application is OOD specific and can thus be easily ported to other deployment scenarios.

Fig. 6.2 describes the actions taken by DALiuGE elements when submitting a graph through open on demand. Importantly, the physical graph deployment is triggered by the user's browser directly, not the machine hosting the translator.

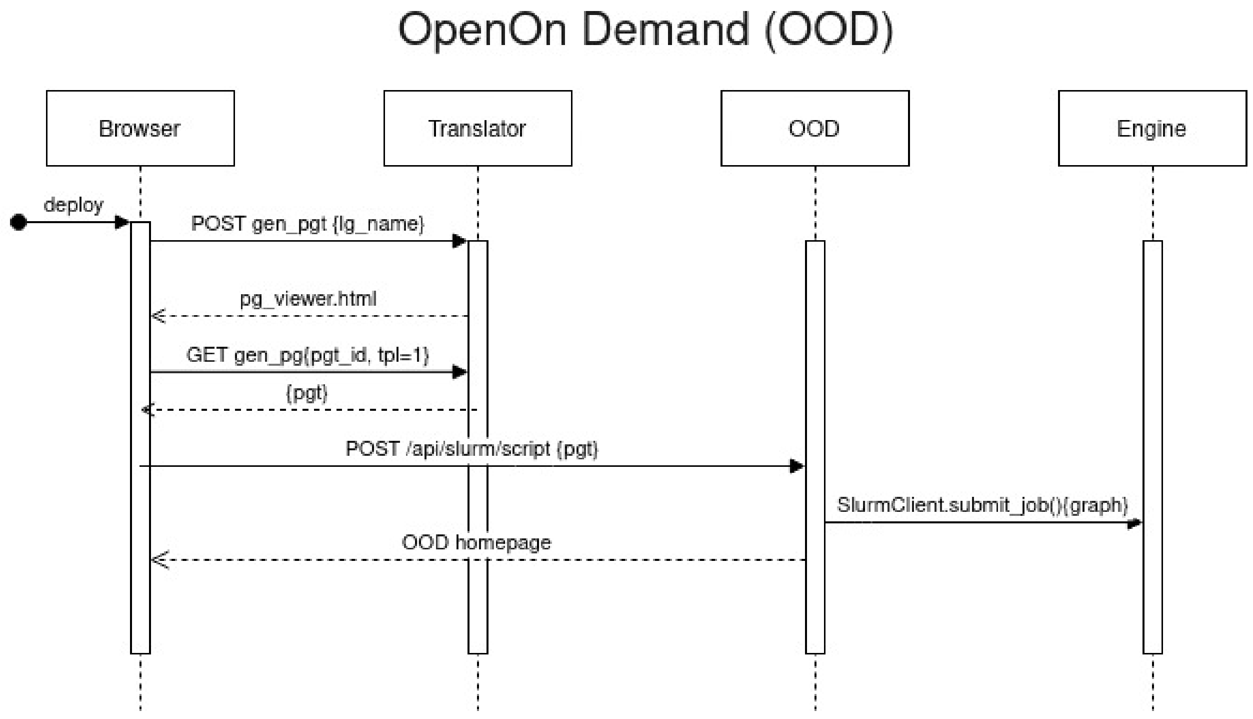


Fig. 6.2: Sequence diagram of graph deployment in OOD envrionment.

6.1.3 Direct Deployment

It is of course possible to submit graphs to DALiuGE managers without additional runtime environments. The manager and translator components can be docker images or raw processes. We currently support two methods for submitting graphs in this scenario.

Server

The server deployment option assumes the machine hosting the translator can communicate with the manager machines freely. Fig. 6.3 presents a sequence diagram outlining the communication between the different components in this case.

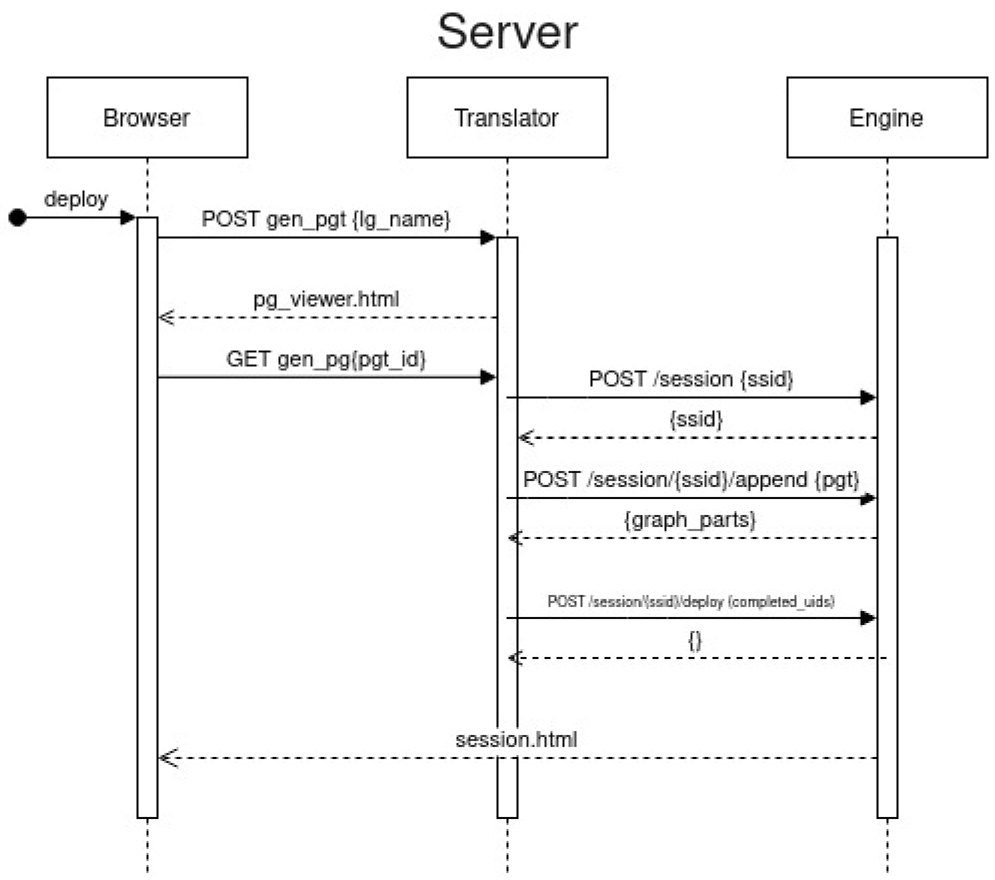


Fig. 6.3: Sequence diagram of direct graph deployment.

Browser

Browser-based deployment is useful in the case where only a user's machine can communicate with engine instances but the translator cannot (as is often the case with an externally hosted translator process). The browser in this case drives execution and submits the graph directly to the manager nodes. Fig. 6.4 presents a sequence diagram outlining the communication between the different components in this case. Conceptually this is similar to how the OpenOnDemand deployment works, but targeting direct graph deployment rather than slurm job submission.

N.B. Cross-Origin Resource Sharing (CORS) may return some interesting responses. If running all machines locally, make sure that your host descriptions in EAGLE and the translator are 'localhost'.

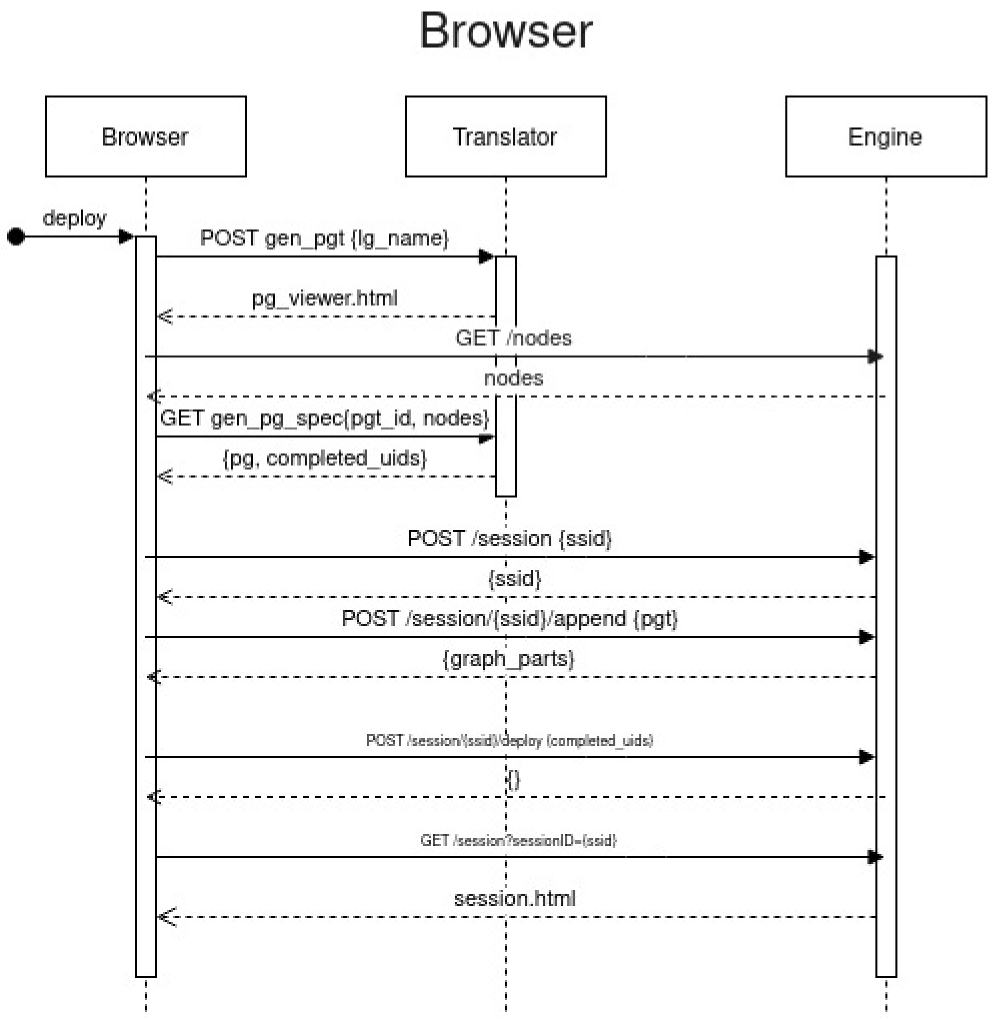


Fig. 6.4: Sequence diagram of restful graph deployment.

6.1.4 Deployment with Kubernetes/Helm (Experimental)

Kubernetes is a canonical container orchestration system. We are building support to deploy workflows as helm charts which will enable easier and more reliably deployments across more computing facilities. Multi-node kubernetes clusters are now supported to get started see start_helm_cluster.py for an example usage. Your environment will need have *kubectl* properly configured to point to your desired cluster. See daliuge-k8s/README.md for a more detailed setup guide.

Fig. 6.5 describes the actions taken by DALiuGE elements when submitting a graph through helm. Importantly, there is (currently) no return to the browser indicating success or failure of the submission or job. The user will need to monitor the k8s environment directly.

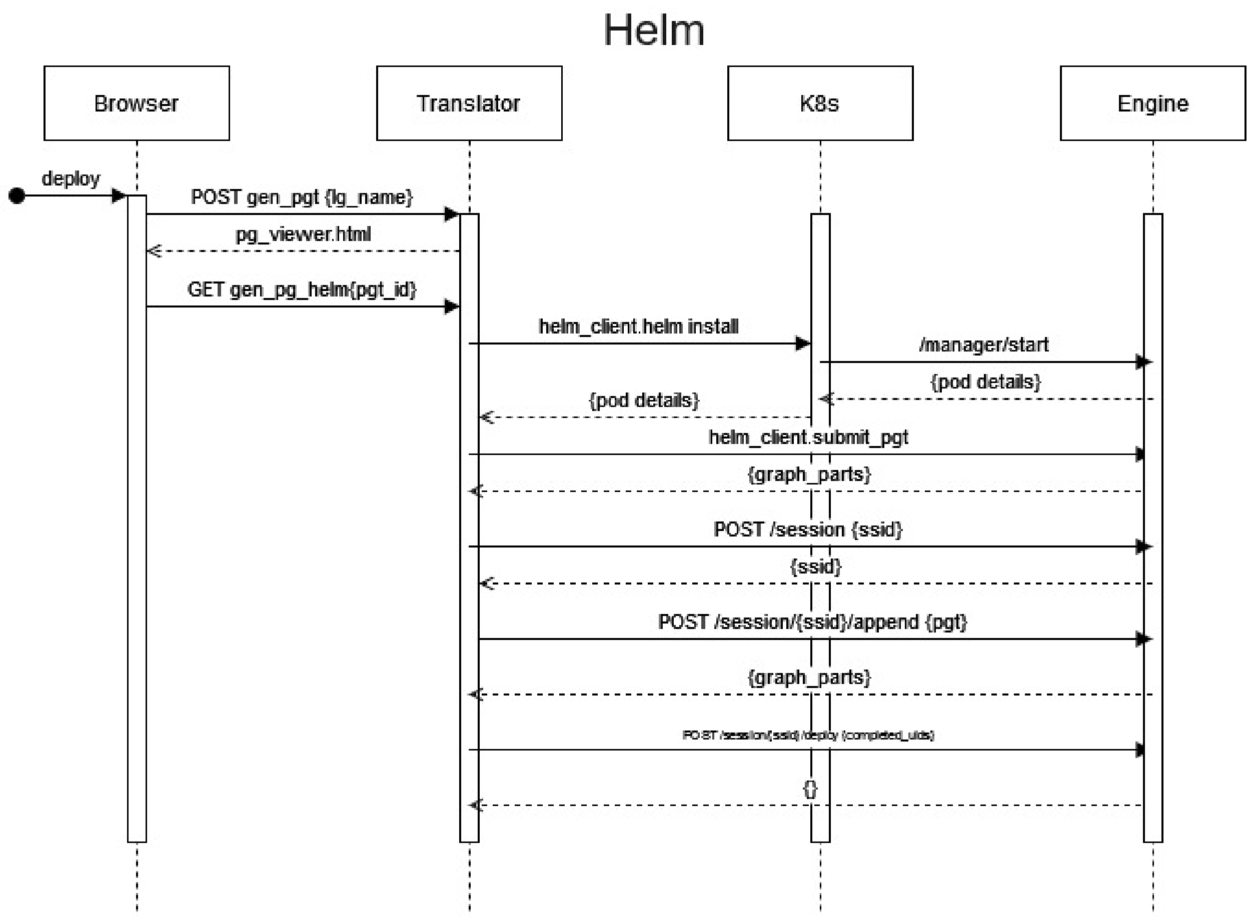


Fig. 6.5: Sequence diagram of graph deployment in helm environment.

6.2 Component Deployment

6.2.1 Docker components

DALiuGE is a workflow development and management system and a workflow execution framework. Workflows rely on components and algorithmic code to perform the actual data reduction. The DALiuGE system does include only a few basic components, everything else needs to be provided and made available to the system externally. The JSON based component descriptions are being used by EAGLE and the translator, the engine needs access to the actual executable code. The most straight forward way to give the DALiuGE engine access to code is to refer to docker images. The engine will pull the images, if not available already and execute them internally as containers. This works even if the DALiuGE managers are launched as docker containers themselfes. Currently we are only supporting docker containers as workflow components. We have tested running the managers as Singularity containers and they internally can still launch docker containers. DALiuGE allows a quite flexible configuration of docker components and the way they are executed. However, there are still a number of restrictions:

(1) Memory Data Components can't be used directly as input or output of Docker components. However, it is possible to use Plasma/Flight as a shared memory mechansim.

(2) Care has to be taken when using files to exchange data between docker components and other components. In particular any usage of absolute path names is quite tricky to get working and requires cross-mounting of additional volumes. Although this is possible it is not recommended. The DALiuGE workspace directory is mounted by default in the container components as well.

(3) By default the docker containers are started with the same user/group ids as the user running the engine.

Note that it is not recommended to pack big external packages together with DALiuGE in a single image. We are using the slimtoolkit to minimize the size of our docker images and recommend doing this for component images as well.

6.2.2 Python components

Components written in Python provide direct access to the whole DALiuGE engine runtime. They can use direct remote procedure calls and memory sharing even across multiple compute nodes. By default the engine is configured to use the multiprocessing module to launch the *application code* of the components using a maximum number of processes equal to the number of physical cores available on the computer. If there are more components than cores, then they are executed in serial. More advanced Python components, which are not restricted by the Python Global Interpreter Lock (GIL) don't really need this mechanism. Memory data components will automatically switch to use shared memory blocks between those processes. Note that the *component code* will still run in a single process together with the node manager. In the future, in order to minimize side effects, we might entirely switch to using separate processes for the execution of application code.

In order to be able to use Python components, it must be possible for the engine to import the code and thus it must be accessible on the PYTHONPATH at runtime. By default the engine is configured to add the directory $DLG_ROOT/code to the PYTHONPATH and thus users can install their code there using a command like:

```
docker exec -ti daliuge-engine bash -c "pip install --prefix=\$DLG_ROOT/code dlg_example_
↪cmpts"
```

Please note that the '' character is required for this to work correctly. In the case of running DALiuGE in docker containers $DLG_ROOT is mounted from the host and thus also the subdirectory code is visible directly on the host. In a typical HPC deployment scenario that directory will be on the user's home directory, or a shared volume, visible to all compute nodes.

CHAPTER

SEVEN

GRAPH DEVELOPMENT

This section describes the different ways users can develop workflows (either Logical or Physical) to work with DALiuGE.

As explained in *Graphs*, DALiuGE describes computations in terms of Directed Graphs. Two different classes of graphs are used in the DALiuGE workflow development:

1. *Logical Graphs*, a high-level, compact representation of the application logic. Logical Graphs are directed graphs, but not acyclic.
2. *Physical Graphs*, a detailed description of each individual processing step. Physical Graphs are Directed Acyclic Graphs (DAG)

When submitting a graph for execution, the DALiuGE engine expects a physical graph. Therefore a logical graph needs to be first translated into a physical graph before submitting it for execution. The individual steps that occur during this translation process are detailed in *Translation*.

The following graph development techniques are available for users to creates graphs and submit them for execution:

1. *Use the Logical Graph Editor EAGLE* to create a logical graph, which can then be translated into a physical graph.
2. Manually, or automatically, *create a Physical Graph from scratch*.
3. *Use the delayed function* to generate a physical graph.

7.1 Using the Logical Graph Editor EAGLE

Please refer to the EAGLE documentation for detailed information. When using the EAGLE graph editor the translator and engine levels are not really exposed to the user, thus in the following we will describe a few examples of how to directly generate a graph.

7.2 Directly creating a Physical Graph

In some cases using EAGLE is not possible or not the preferred way of working.

In these cases, developing a Physical Graph directly is still possible. One example where this approach is used, are the DALiuGE engine tests, more specifically in the subdirectory `daliuge-engine/test/apps`. As can be seen there the graph is constructed directly in Python, by using high-level class methods to instantiate application and data nodes and then adding inputs and outputs, or producers and consumers, for applications and data nodes, respectively. Note that you only have to add either a *consumer* to a data node, or the equivalent *input* to the application node. Once the whole graph is constructed in that way, it can be executed directly using using the utility method `droputils.DROPWaiterCtx`. For smaller graphs this is a perfectly valid approach, but it is quite tedious when it comes to larger scale graphs.

The test `daliuge-engine/test/manager/test_scalability` contains an example of how to generate a big graph using higher level functions. However, this approach is only feasible for large but low complexity graphs. Since the constructs (e.g. Scatter, Gather, Loop) are also exposed as classes, they can also be used in the same way as normal apps to construct more complex graphs.

7.2.1 Simple Hello World graph

Like every software framework project we need to describe a Hello World example. This one is straight from the DALiuGE test in `daliuge-engine/test/apps/test_simple.py`:

```
1 from dlg.apps.simple import HelloWorldApp
2 from dlg.drop import FileDROP
3
4 h = HelloWorldApp('h', 'h')
5 f = FileDROP('f', 'f')
6 h.addOutput(f)
7 f.addProducer(h)
8 h.execute()
```

Let's look at this in detail. Lines 1 and 2 import the HelloWorldApp and the FileDROP classes, respectively, both of them are part of the DALiuGE code base. Line 4 instanciates an object from the HelloWorldApp class and assigns an object ID (oid) and a unique ID (uid) to the resulting object. In our example both of them are simply the string `'h'`. We then instantiate the FileDROP with `oid = uid = 'f'` in line 5. In line 6 we add the instance of the FileDROP (f) as an output to the HelloWorldApp drop (h). In line 7 we add the HelloWorldApp drop (h) as a producer for the FileDROP (f). NOTE: It would have been sufficient to have either line 6 or line 7, but just to show the calls we do both here (and it does not break things either). Finally in line 8 we call the execute method of the HelloWorldApp (h). This will trigger the execution of the rest of the graph as well. Note that there was no need to care about any detail at all. In fact it is not even obvious whether anything happend at all when executed. In order to check that let's have a look where the file had been written to:

```
in [1] print(f.path, f.size)
/tmp/daliuge_tfiles/f 11
```

Means that there is a file with name f and a size of 11 bytes:

```
in [2] print(len('Hello World'))
11
in [3] !cat $f.path
Hello World
```

Seems to be what is expected!

7.2.2 Parallel Hello World graph

Now that was fun, but kind of boring. DALiuGE is all about paralellism, thus we'll add a bit of that:

```
from dlg.apps.simple import HelloWorldApp, GenericScatterApp
from dlg.drop import FileDROP, InMemoryDROP
from dlg.droputils import DROPWaiterCtx
import pickle

m0 = InMemoryDROP('m0','m0')
```

(continues on next page)

(continued from previous page)

```
7  s = GenericScatterApp('s', 's')
8  greets = ['World', 'Solar system', 'Galaxy', 'Universe']
9  m0.write(pickle.dumps(greets))
10 s.addInput(m0)
11 m = []
12 h = []
13 f = []
14 for i in range(1, len(greets)+1, 1):
15    m.append(InMemoryDROP('m%d' % i, 'm%d' % i))
16    h.append(HelloWorldApp('h%d' % i, 'h%d' % i))
17    f.append(FileDROP('f%d' % i, 'f%d' % i))
18    s.addOutput(m[-1])
19    h[-1].addInput(m[-1])
20    h[-1].addOutput(f[-1])
21 with DROPWaiterCtx(None, f, 1):
22    m0.setCompleted()
```

This example is a bit more busy, thus let's dissect it as well. In the import section we import a few more items, the GenericScatterApp and the InMemoryDROP as well as the pickle module. In lines 5 and 6 we instantiate an InMemoryDROP and a GenericScatterApp respectively. Line 7 just prepares the array of strings, called *greets* to be used as greeting strings. In line 7 we push that array into the memory drop *m0*. Line 8 adds *m0* to the scatter app as input. Lines 9,10 and 11 just initialize three lists and in line 12 we start a loop for the number of elements of *greets*. This loop is essentially the main construction of the rest of the graph as well as keeping all the drop objects in the three lists *m*, *h* and *f* (lines 13, 14 and 15). Each element of *greets* will be placed into a separate memory drop by the GenericScatterApp (line 16). Each of those memory drops will trigger a separate HelloWorldApp drop (line 17), which in turn will write to a separate file drop (line 18). Line 19 is using the utility *DROPWaiterCtx* method, which sets up the event subscription mechanism between the various drops in the graph. Finally in line 20 we trigger the execution by changing the status of the initial memory drop *m0* to 'COMPLETE'.

This should now have generated four output files in the default DALiuGE output directory `/tmp/daliuge_tfiles`. If you copy and paste the above script into a file called `parallelHelloWorld.py` and execute it using `ipython -i parallelHelloWorld.py` you can check the content of the files with following commands:

```
In [1]: for fd in f:
...:     fp = fd.path
...:     !cat $fp
...:     print()
```

This should produce the output:

```
Hello World
Hello Solar system
Hello Galaxy
Hello Universe
```

Note that all of the above is still limited to execution on a single node. In order to use the distributed functionality of the DALiuGE system it is still required to use the JSON version of graphs, which in turn lead to the individual drops to be instantiated on the assigned compute nodes. That is also when the I/O transparency suddenly makes sense, because DALiuGE will make sure that the reads and writes are translated into remote reads and writes where and when required. Producing a distributed JSON graph programmatically is possible, albeit a bit tedious, since it essentially requires to construct the JSON representation of the graph and then submit it to the DALiuGE island manager. This is shown in more detail in the file `daliuge-engine/test/manager/test_scalability.py`.

7.3 Using `dlg.delayed()`

DALiuGE ships with a Dask emulation layer that allows users write code like they would for using under Dask, but that executes under DALiuGE instead. In Dask users write normal python code to represent their computation. This code is not executed immediately though; instead its execution is *delayed* by wrapping the function calls with the `delayed` Dask function, until a final `compute` call is invoked, at which point the computation is submitted to the Dask runtime agents. These agents execute the computation logic and return a result to the user.

To emulate Dask, DALiuGE also offers a `delayed` function (under `dlg.delayed`) that allows users to write normal python code. The usage pattern is exactly the same as that of Dask: users wrap their function calls with the `delayed` function, and end up calling the `compute` method to be obtain the final result.

Under the hood, DALiuGE returns intermediate placeholder objects on each invocation to `delayed`. When `compute` is invoked, these objects are used to compute a Physical Graph, which is then submitted to one of the Drop Managers for execution. DALiuGE doesn't have the concept of returning the final result back to the user. In order to imitate this, a final application is appended automatically to the Physical Graph before submission. This final application allows the `compute` function to connect to it. Once this final application receives the final result of the Physical Graph it then sends it to the `compute` function, who presents the result to the user.

- genindex
- search

CHAPTER

EIGHT

DALIUGE COMPONENT DEVELOPERS GUIDE

Welcome to the DALiuGE Component Developers Guide (DCDG).

The DALiuGE workflow graph execution framework enables an almost complete separation of concerns between graph development and component development. This guide specifically addresses the concern of component development. Please refer to the DALiuGE *Introduction* for a top-level introduction to the entire system, and read our overview paper.

NOTE: The DCDG is work in progress!

8.1 Introduction to Component development

8.1.1 What are *Components*?

Nomenclature

The following chapters and sections will use some terms in a specific meaning. The definitions are given below.

1. Component: *Component* refers to a DALiuGE compliant implementation of some functionality used in the execution of a workflow. A component consists of the DALiuGE interface wrapper and the code implementing the desired functionality. In some cases the actual functional code is not integrated with the interface, but just executed by the interface. There are three main types of components:
   - Application Component: DALiuGE interface wrapper and application code.
   - Data Component: DALiuGE interface wrapper around an I/O channel. Examples are standard files, memory, S3 and Apache Plasma.
   - Service Component: A *Service Component* is a special component, providing access to services like a database or some other client/server system used within a workflow.
2. Construct: A *Construct* is a complex *Component*, which may contain other *Components*.
3. Node: *Graph Node* or *Palette Node* refers to a JSON representation of a *Component* in a DALiuGE graph or palette.
4. Drop: *Drop* is a DALiuGE specific term used to describe instances of data, application or service components at execution time. In general developers don't have to dive into the Drop level.

In practice the component interface wrapper code is written in Python. DALiuGE provides generic wrappers and base classes to make the development of components more straight forward and hide most of the DALiuGE specifics. In some cases the generic wrappers can be used directly to develop functioning Graph and Palette Nodes using EAGLE, without writing any code. Examples are bash nodes (*Bash Components*).

Seperation of concerns

The DALiuGE system has been designed with the separation of concerns in mind. People or groups of people can work independently on the development of

1. the logic of a workflow (graph),
2. the detailed definition of a Node, and a collection of nodes (Palette)
3. the interface Component code and
4. the actual functional implementation of the required algorithm of a component.

In fact it is possible to create logical workflows and run them, without having any substantial functional code at all. On the opposite side it is also possible to develop the functional software without considering DALiuGE at all. This feature also allows developers to write wrapper components around existing software without the need to change that package. Whatever can be called on a *NIX bash command line, in a docker container, or can be loaded as a python function can also run as part of a DALiuGE workflow.

The Component code has been designed to be as non-intrusive as possible, while still allowing for highly optimized integration of code, down to the memory level. With a bit of care developers can fairly easily also run and test each of these layers independently.

Integration of Layers

The DALiuGE system consists of three main parts:

1. EAGLE for graph development
2. Translator to translate logical graphs to physical graphs and partition and statically schedule them.
3. Execution Engine to execute physical graphs on small or very large compute clusters.

All three of them are essentially independent and can be used without the others running. The EAGLE visual graph editor is a web application, which deals with *Nodes*, i.e. with JSON descriptions of *Components*. It also allows users to define these descriptions starting from node templates. Users can group and store sets of defined nodes as *palettes*. The nodes of one or multiple palettes are then used to construct a workflow graph. Graphs and Palettes are stored as JSON files and they are essentially the same, except for the additional arrangement and visual information in a graph file. A graph can be stored as a palette and then used to create other graphs.

In addition to constructing the nodes manually in EAGLE it is also possible to generate the JSON description of the nodes in a palette from the component interface code by means of special in-line doxygen documentation tags. This procedure is described in detail in *Component Description Generation*. The idea here is that the developers of the component can stay within their normal development environment and just provide some additional in-line code documentation to allow people to use their new component in their workflows.

8.1.2 Component development

In particular if people already have existing Python code or large libraries, we are now recommending to first try our new stand-alone tool dlg_paletteGen), which enables the automatic generation of DALiuGE compatible component descriptions from existing code. The tool parses the code and generates a palette containing the classes, class methods and functions found in the code. It does not require any code changes at all. However, for best usability of the resulting components good docstrings and Python type hinting is highly recommended. The tool supports three standard docstring formats: Google, Numpy and ReST. Also for new components and in particular for all these small little functions, which will be required to perform exactly the tasks you need, we now recommend to simply put them in a .py file and let the tool do the rest. Please refer to the tool documentation for more information.

Simple bash and python application components as well as data components can be developed completely inside EAGLE and, since EAGLE interfaces with gitHub and gitLab, it even provides some kind of development workflow. However, for more complex and serious component development it is strongly recommended to use the component development template we are providing. The template covers application and data components and provides everything to get you started, including project setup, testing, format compliance, build, documentation and release, continuous integration and more. Although it is meant to be used to develop a whole set of components, it is quite useful even for just a single one. We are still actively developing the template itself and thus a few things are still missing:

1. Installation into the DALiuGE runtime is possible, but not the way we would like to have it.
2. Automatic palette generation is not yet integrated.

Please note that most of the Components Developers Guide is based on using the template.

8.2 Template Primer

We are providing a GitHUB component development template. The template covers application and data components and provides everything to get you started, including project setup, testing, format compliance, build, documentation and release, continuous integration and more. Although it is meant to be used to develop a whole set of components, it is quite useful even for just a single one. We are still actively developing the template itself and thus a few things are still missing, or have rough edges.

8.2.1 Using the GitHUB template

First of all, in order to use the template functionality you need to have an account and be logged in to GitHub. The template GitHUB page contains usage information and the GitHUB documentation contains general information about template usage as well. Here we provide some additional information about our template in particular. When generating a DALiuGE component project from the template, this will generate a complete GitHUB project structure including code and documentation templates as well as a Makefile containing targets for testing, liniting, formatting and installation. The creation of the new project is triggering a number of GitHub actions, which need to be finished, *before* cloning the project.

After clicking on the **Use this template** button you will be presented with a screen like the one shown below.

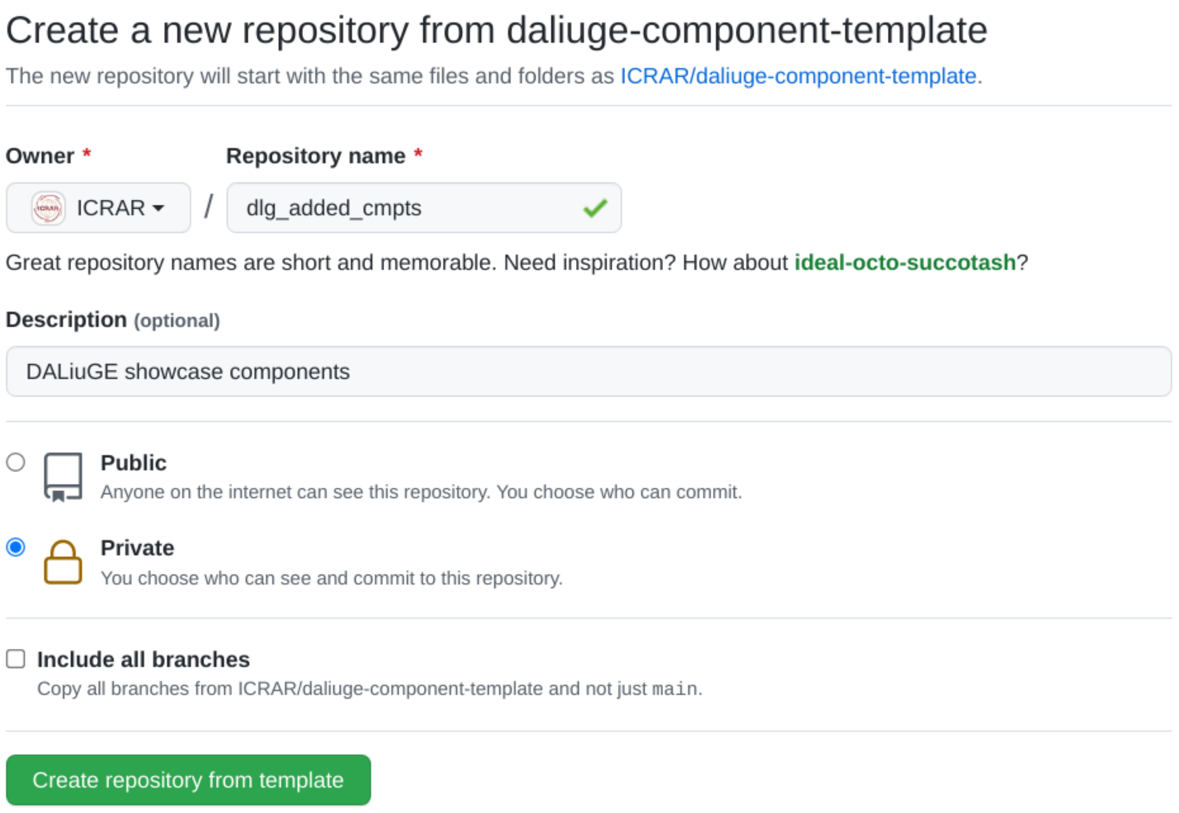
You need to select an owner and enter a repository name. Github will check whether that name is available and put the green tickmark behind the name. You can select to make the repo private or public straight away. Once done, click **Create repository from template**. This will go away and generate a new repository with the owner and name you have specified. It is not quite usable just yet, since there are a number of GitHub actions executed, one of which is going through the files and perform some magic to make the project actually look like yours, rather than just a copy of the template. You can watch the progress by clicking on the **Actions** tab just underneath the project name in the GitHub page. This should look like:

In addition you can click on one of the running actions and get a bit more detail:

Once that is all completed the project will be committed again and then it is ready to be cloned to your local machine. Just click on the green **Code** button in the main project page and then on the copy icon next to the URL:

Once cloned, please make sure you have a daliuge compatible Python version as your default. Compatible versions are the 3.7 and 3.8 series. Make also sure that you are not already inside a virtualenv. This would very likely screw things up. You should then execute

```
make virtualenv
source .venv/bin/activate
make install
```



Create a new repository from daliuge-component-template

The new repository will start with the same files and folders as ICRAR/daliuge-component-template.

Owner * ICRAR / Repository name * dlg_added_cmpts

Great repository names are short and memorable. Need inspiration? How about ideal-octo-succotash?

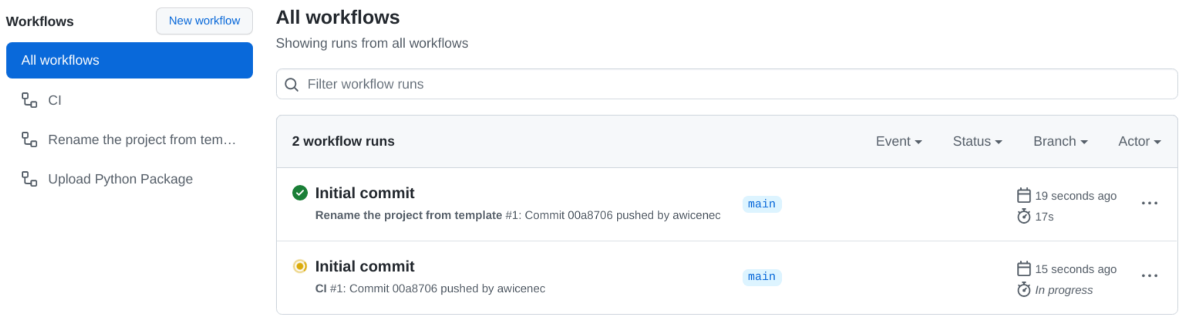
Description (optional)

DALiuGE showcase components

( ) Public
Anyone on the internet can see this repository. You choose who can commit.

[✓ Private]
You choose who can see and commit to this repository.

☐ Include all branches
Copy all branches from ICRAR/daliuge-component-template and not just `main`.

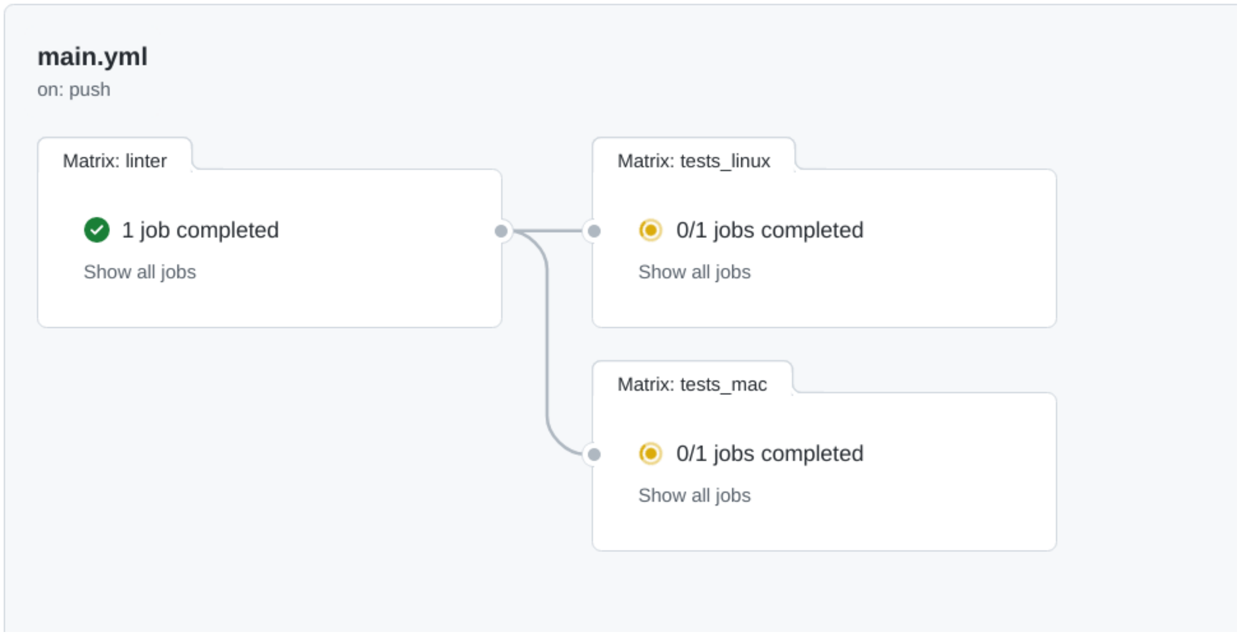


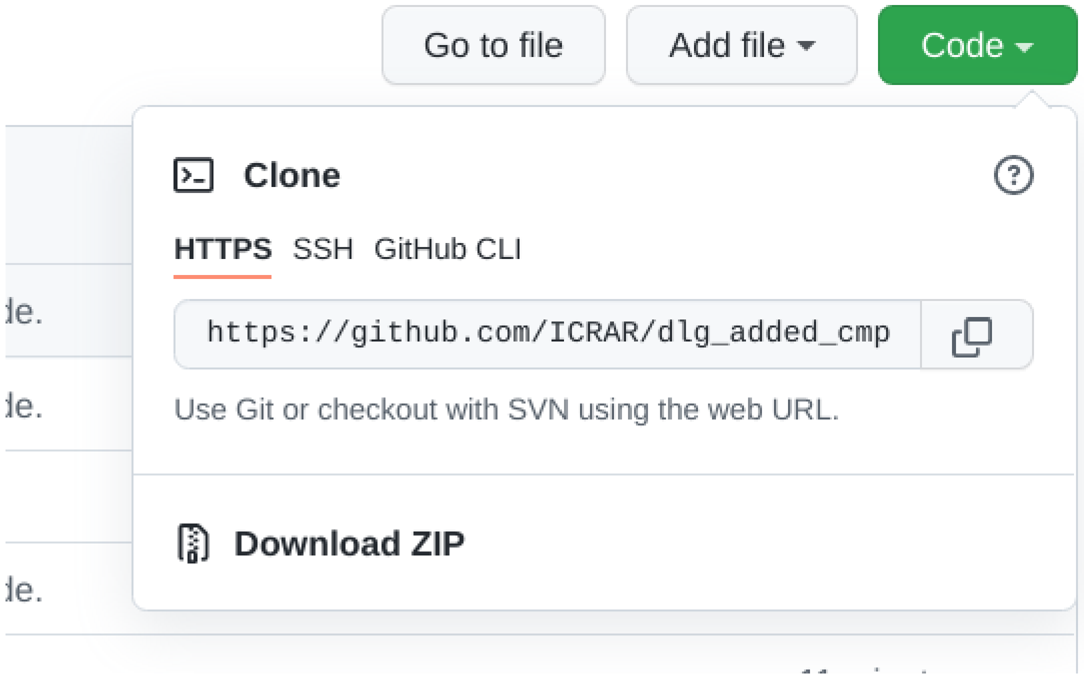
All workflows

Showing runs from all workflows

2 workflow runs

✓ Initial commit Rename the project from template #1: Commit 00a8706 pushed by awicenec	main	19 seconds ago 17s	…
● Initial commit CI #1: Commit 00a8706 pushed by awicenec	main	15 seconds ago In progress	…



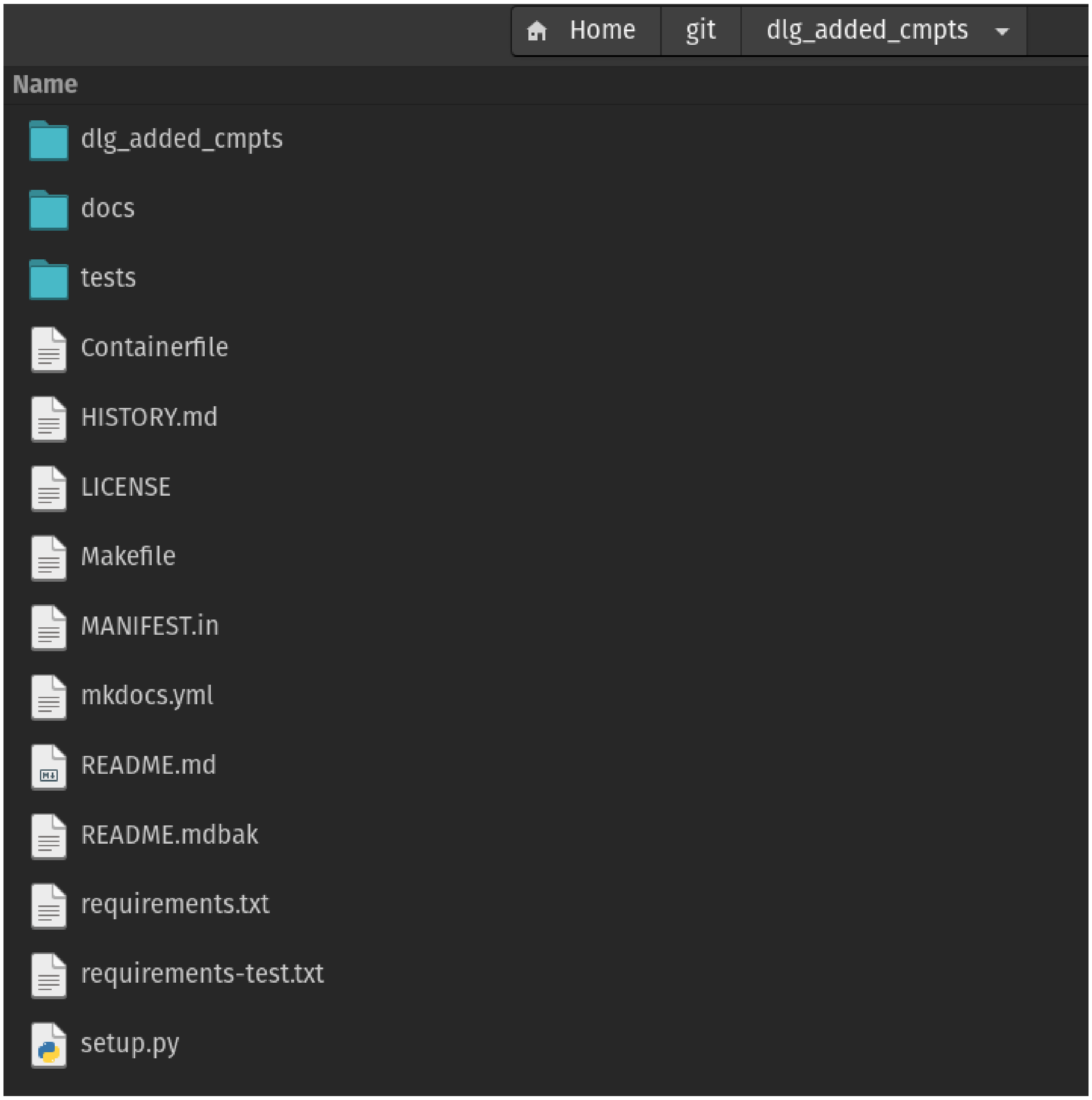


```
https://github.com/ICRAR/dlg_added_cmp
```

Use Git or checkout with SVN using the web URL.

8.2.2 Component project directory structure

After using the template and cloning your new project you will have a directory structure like the one in the figure below. The `dlg_added_cmpts` directory will be different, but else this is what you should see.



```
dlg_added_cmpts
docs
tests
Containerfile
HISTORY.md
LICENSE
Makefile
MANIFEST.in
mkdocs.yml
README.md
README.mdbak
requirements.txt
requirements-test.txt
setup.py
```

1. The `.github` directory contains the standard GitHUB config and history files, but also some GitHUB action definitions, which will be exectuted by GitHUB whenever your project is pushed. There is one action, which is executed only once, when the new project is created and that will peform some global re-naming and other things, so that you don't have to do that yourself.
2. The `docs` directory contains just a stub for your documentation to be provided together with the components.
3. The project_name directory will be renamed to whatever you called your component project at creation time. It is a Python module and thus contains an `__init__.py` file.

4. The `tests` directory contains the *conftest.py* ` file and a `test_components.py` file, which contains tests passing with the code stubs in the (renamed!) `dlg_added_cmpts` directory.
5. The file `.gitignore` contains a whole set of default files to be excluded from a commit.
6. The `Contributing.md` file is just a bit a information on how to contribute to the template development.
7. The `Containerfile` is a stub for a build file for a container (e.g. docker)
8. The HISTORY.md file is automatically maintained when doing a release.
9. The `LICENSE` file contains *The Unlicense* text. Modify as you see fit for your component code. Since the code is not delivered together with the DALiuGE system there is no direct restriction from that side.
10. The MANIFEST.in file just pulls together a number of other files in the same directory.
11. The `README.md` file should be modified to reflect your component project.
12. The `mkdocs.yml` file is the configuration for the production of the documentation.
13. The `requirements-test.txt` file contains a list of all required packages for testing the components. This will be used when executing `make install`.
14. The `requirements.txt` file contains a list of all required packages for testing the components. This will be used when executing `pip install .`.
15. The file `setup.py` is used when installing the component package using `pip install .`.

Now you are all set and can start coding. In the following sections we will describe how to develop a simple component in this environment and how to get your new components into the system.

- genindex
- search

8.3 DALiuGE *Application* Component Developers Guide

This chapter describes what developers need to do to write a new application component that can be used as an Application Drop during the execution of a DALiuGE graph.

Detailed instructions can be found in the respective sections for each type of components. There are also separate sections describing integration and testing during component development. As mentioned already, for more complex and serious component development we strongly recommend to use the component development template we are providing, please refer to chapter *Template Primer* for more details. Most of the following sub-sections of this documentation are based on the usage of the template.

NOTE: The DALiuGE Component Developers Guide is work in progress!

8.3.1 Bash Components

`BashShellApp` components are probably the easiest to implement and for simple ones it is possible to do all the 'development' in EAGLE.

'Hello World' in Bash through EAGLE

Steps

- Open EAGLE (e.g. https://eagle.icrar.org) and create a new graph (Graph –> New –> Create New Graph)
- Drag and drop a 'Bash Shell App' from the 'All Nodes' palette on the right hand panel onto the canvas.
- Click on the 'Bash Shell App' title bar in the Inspector tab on the right hand panel. This will open all additional settings below.
- First change the 'Name' field of the app in the 'Display Options' menu. Call it 'Hello Word'. Once you leave the entry field also the black title bar will reflect that new name.
- Now change the description of the app in the 'Description' menu. Maybe you write 'Simple Hello World bash app'.
- now go down to the 'Component Parameters' menu and enter the bash command in the 'Command' field:

  ```
  echo "Hello World"
  ```

- Now save your new toy graph (Graph –> Local Storage –> Save Graph).

Please note that the *Hello World* example is also described (with videos) as part of the EAGLE documentation.

That should give you the idea how to use bash commands as DALiuGE components. Seems not a lot? Well, actually this is allowing you to execute whatever can be executed on the command line where the engine is running as part of a DALiuGE graph. That includes all bash commands, but also every other executable available on the PATH of the engine. Now that is a bit more exciting, but the excitement stops as soon as you think about real world (not Hello World) examples: Really useful commands will require inputs and outputs in the form of command line parameters and files or pipes. This is discussed in the *Advanced Bash Components* chapter.

Verification

Do we believe that this is actually really working? Well, probably not. Thus let's just translate and execute this graph. Note that the graph has neither an input nor an output defined, thus there is nothing you could really expect from running it. However, the DALiuGE engine is pretty verbose when run in debug mode and thus we will use that to investigate what's happening. The following steps are very helpful when it comes to debugging actual components.

Assuming you have a translator and an engine running you can actually translate and execute this, pretty useless, graph. If you have the engine running locally in development mode, you can even see the output in the session log file:

```
cd /tmp/dlg/logs
ls -ltra dlg_*
```

The output of the ls command looks like:

```
-rw-r--r-- 1 root root 1656 Sep 14 16:46 dlg_172.17.0.3_Diagram-2021-09-14-16-41-283_
↪2021-09-14T08-46-17.341082.log
-rw-r--r-- 1 root root 6991 Sep 14 16:46 dlg_172.17.0.3_Diagram-2021-09-14-16-41-284_
↪2021-09-14T08-46-52.618798.log
-rw-r--r-- 1 root root 6991 Sep 14 16:47 dlg_172.17.03_Diagram-2021-09-14-16-41-284_2021-
↪09-14T08-47-28.890072.log
```

There could be a lot more lines on top, but the important one is the last line, which is the log-file of the session last executed on the engine. Just dump the content to the screen in a terminal:

```
cat dlg_172.17.03_Diagram-2021-09-14-16-41-284_2021-09-14T08-47-28.890072.log
```

Since the engine is running in debugging mode there will be many lines in this file, but towards the end you will find something like:

```
2021-09-14 08:47:28,912 [ INFO] [      Thread-62] [2021-09-14] dlg.apps.bash_shell_app#_
↪run_bash:217 Finished in 0.006 [s] with exit code 0
2021-09-14 08:47:28,912 [DEBUG] [      Thread-62] [2021-09-14] dlg.apps.bash_shell_app#_
↪run_bash:220 Command finished successfully, output follows:
==STDOUT==
Hello World

2021-09-14 08:47:28,912 [DEBUG] [      Thread-62] [2021-09-14] dlg.manager.node_manager
↪#handleEvent:65 AppDrop uid=2021-09-14T08:46:48_-1_0, oid=2021-09-14T08:46:48_-1_0␣
↪changed to execState 2
2021-09-14 08:47:28,912 [DEBUG] [      Thread-62] [2021-09-14] dlg.manager.node_manager
↪#handleEvent:63 Drop uid=2021-09-14T08:46:48_-1_0, oid=2021-09-14T08:46:48_-1_0␣
↪changed to state 2
```

In addition to the session log file the same information is also contained in the dlgNM.log file in the same directory. That file contains all logs produced by the node manager for all sessions and more, which is usually pretty distracting. However, the name of the session logs are not known before you deploy the session and thus another trick is to monitor the dlgNM.log using the tail command:

```
tail -f dlgNM.log
```

When you now deploy the graph again and watch the terminal output, you will see a lot of messages pass through.

Treatment of Command Line Parameters

DALiuGE has multiple ways to pass command line parameters to a bash command. The same feature is also used for Docker, Singularity and MPI components. In order to facilitate this DALiuGE is using a few reserved component parameters:

- command: The value is used as the command (utility) to be executed. It is possible to specify more complex command lines in this value, but the end-user needs to know the syntax.
- command_line_arguments: Similar to the command, but the value only contains any additional arguments. Again the end-user needs to know all the details.
- Arg00 - Arg09: (.. deprecated:: use applicationArgs instead, only kept for backwards compatibility)
- applicationArgs: This is serialized as separate dictionary in JSON and every applicationParam is one entry where the key is the parameter name. This is the most recent way of defining and passing arguments on the command line and allows the developer to define every single argument in detail, including some description text, default value, write protection and type specification/verification. EAGLE displays the complete set and allows users to spicify and modify the content. DALiuGE will process and concatenate all of the parameters and attach them to the command line. In order to enable the user/developer to control the behaviour of the processing there are two additional parameters:
- argumentPrefix: The value is prepended to each of the parameter names of the applicationArgs. If not specified, the default is '–'. In addition, if argumentPrefix=='–' and an argument name is only a single character long, the argumentPrefix will be changed to '-'. This allows the construction of POSIX compliant option arguments as well as short argument names.

- paramValueSeparator: The value is used to concatenate the argument name and the value. The default is ' ' (space). Some utilities are using a syntax like 'arg=value'. In that case this parameter can be set accordingly.
- input_redirection: the value will be prepended to the command line (cmd) using 'cat {value} > '.
- output_redirection: the value will be appended to the command line (cmd) using '> {value}'.

Not all of them need to be present in a component, only the ones the component developer wants to offer to the user. In particular the applicationArgs have been introduced to support complex utilties which can feature more than 100 arguments (think about tar). If more than one way of specifying arguments is available to an end-user, they can be used together, but the order in which these parts are concatenated might produce unwanted results. The final command line is constructed in the following way (not including the deprecated ArgXX parameters)::

cat {input_redirection.value} > {command.value} {argumentPrefix.value}{applicationArgs.name}{paramValueSeparator}{applic
{command_line_arguments.value} > {output_redirection}

The applicationArgs are treated in the order of appearance. After the construction of the command line, any placeholder strings will be replaced with actual values. In particular strings of the form '%iX' (where X is the index number of the inputs of this component), will be replaced with the input URL of the input with that index (counting from 0). Similarly '%oX' will be replaced with the respective output URLs.

Eventually we will also drop support for the command_line_arguments parameters. However, currently the applicationArgs can't be used to specify positional arguments (just a value) and thus, as a fallback users can still use one the command_line_arguments to achieve that. It should also be noted that for really simple commands, like the one used in the helloWorld example, users can simply specify that in the command parameter directly and ommit all of the others.

Advanced Bash Components

TODO

8.3.2 Python Components

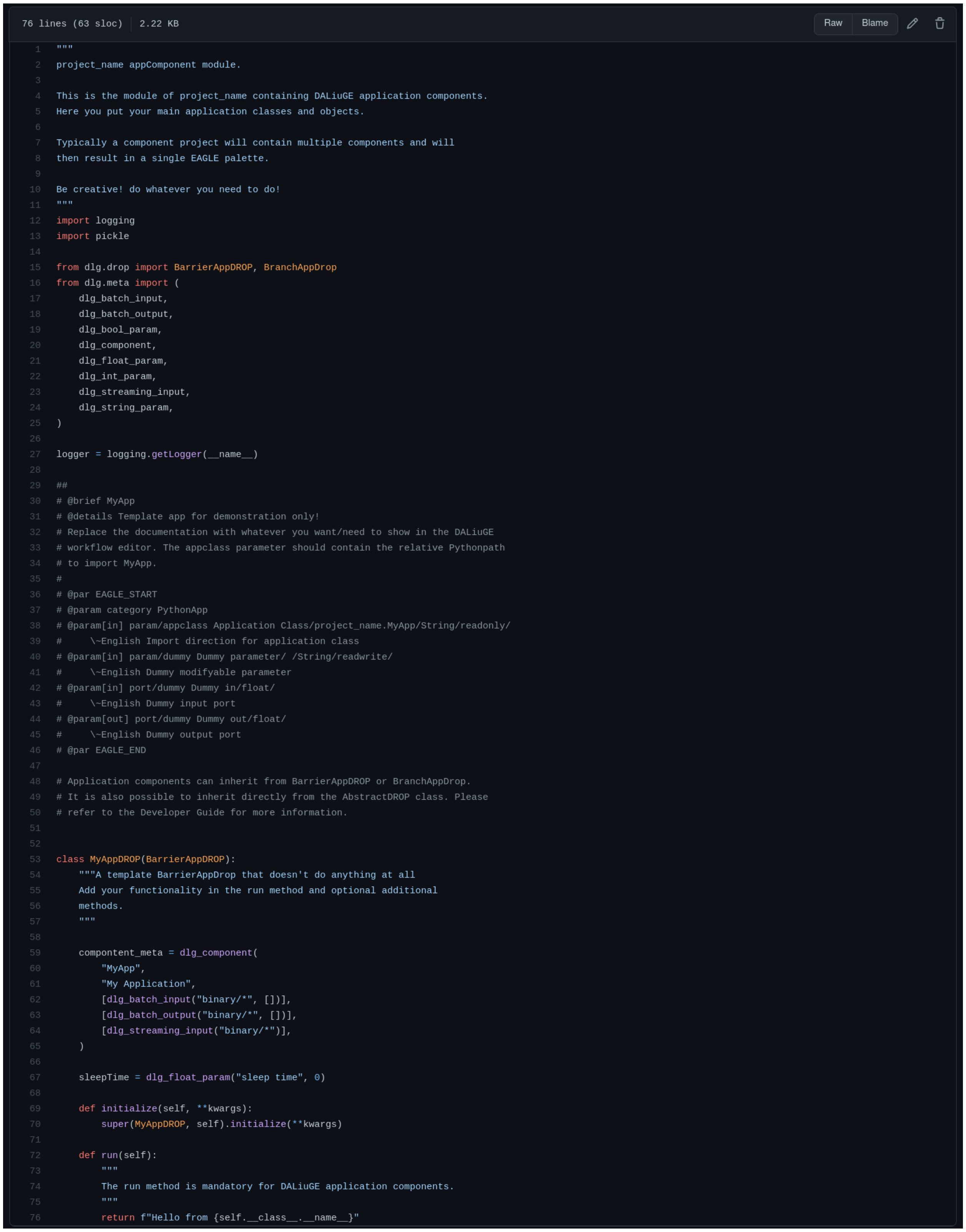
We strongly recommend to use the component development template we are providing, please refer to chapter *Template Primer* for more details. The following is based on the usage of the template.

Change to the sub-directory `my_components` and open the file `app_components.py`:

You will need to modify quite a bit of this file in order to produce a complete template. However, we've tried to make this as easy as possible. The file has three main parts:

1. Generic module level in-line documentation
2. Import section: This will bind the component to the DALiuGE system.
3. Doxygen/Sphinx formatted component documentation: This will be used to generate JSON files compatible with EAGLE and will thus enable people to use your components in the visual graph editor.
4. The actual functionality of a standard Python component is contained in the class MyAppDROP. That in turn inherits from the DALiuGE class BarrierAppDROP.

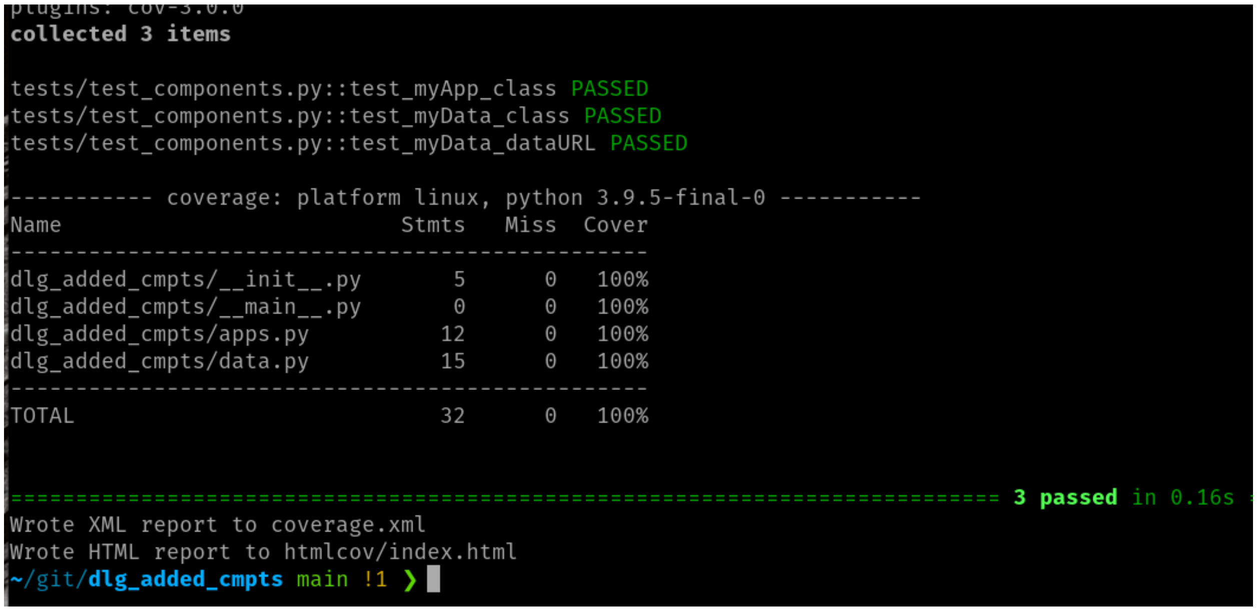
This base class defines all methods and attributes that derived class need to function correctly. This new class will need a single method called `run <dlg.drop.InputFiredAppDROP.run>`,that receives no arguments (except `self`), and executes the logic of the application.



```python
"""
project_name appComponent module.

This is the module of project_name containing DALiuGE application components.
Here you put your main application classes and objects.

Typically a component project will contain multiple components and will
then result in a single EAGLE palette.

Be creative! do whatever you need to do!
"""
import logging
import pickle

from dlg.drop import BarrierAppDROP, BranchAppDrop
from dlg.meta import (
    dlg_batch_input,
    dlg_batch_output,
    dlg_bool_param,
    dlg_component,
    dlg_float_param,
    dlg_int_param,
    dlg_streaming_input,
    dlg_string_param,
)

logger = logging.getLogger(__name__)

##
# @brief MyApp
# @details Template app for demonstration only!
# Replace the documentation with whatever you want/need to show in the DALiuGE
# workflow editor. The appclass parameter should contain the relative Pythonpath
# to import MyApp.
#
# @par EAGLE_START
# @param category PythonApp
# @param[in] param/appclass Application Class/project_name.MyApp/String/readonly/
#     \~English Import direction for application class
# @param[in] param/dummy Dummy parameter/ /String/readwrite/
#     \~English Dummy modifyable parameter
# @param[in] port/dummy Dummy in/float/
#     \~English Dummy input port
# @param[out] port/dummy Dummy out/float/
#     \~English Dummy output port
# @par EAGLE_END

# Application components can inherit from BarrierAppDROP or BranchAppDrop.
# It is also possible to inherit directly from the AbstractDROP class. Please
# refer to the Developer Guide for more information.


class MyAppDROP(BarrierAppDROP):
    """A template BarrierAppDrop that doesn't do anything at all
    Add your functionality in the run method and optional additional
    methods.
    """

    compontent_meta = dlg_component(
        "MyApp",
        "My Application",
        [dlg_batch_input("binary/*", [])],
        [dlg_batch_output("binary/*", [])],
        [dlg_streaming_input("binary/*")],
    )

    sleepTime = dlg_float_param("sleep time", 0)

    def initialize(self, **kwargs):
        super(MyAppDROP, self).initialize(**kwargs)

    def run(self):
        """
        The run method is mandatory for DALiuGE application components.
        """
        return f"Hello from {self.__class__.__name__}"
```

Basic development method

Since the code already implements a sort of a Hello World example we will simply modify that a bit to see how the development would typically work. In the spirit of test driven development, we will first change the associated test slightly and then adjust the component code accordingly to make the tests pass again. First let's see whether the existing tests pass:



```
plugins: cov-3.0.0
collected 3 items

tests/test_components.py::test_myApp_class PASSED
tests/test_components.py::test_myData_class PASSED
tests/test_components.py::test_myData_dataURL PASSED

----------- coverage: platform linux, python 3.9.5-final-0 -----------
Name                          Stmts   Miss  Cover
-------------------------------------------------
dlg_added_cmpts/__init__.py       5      0   100%
dlg_added_cmpts/__main__.py       0      0   100%
dlg_added_cmpts/apps.py          12      0   100%
dlg_added_cmpts/data.py          15      0   100%
-------------------------------------------------
TOTAL                            32      0   100%


============================================================================ 3 passed in 0.16s
Wrote XML report to coverage.xml
Wrote HTML report to htmlcov/index.html
~/git/dlg_added_cmpts main !1 ❯
```

All tests are passing and code coverage in 100%! When you scroll up and look what actually had been done, you will discover that the Makefile has executed flake8, black, mypy, pytest and coverage. Those tools pretty much make sure that the code is in a healthy state and well formatted. In particular black is really helpful, since it actually allows to automatically format the code according to the Python coding standards. When executing `make test` it is only checking, but there is also a `make fmt`, which actually performs the re-formatting.

Test Driven Development workflow

All good! Now change to the tests directory and load the file `test_components.py`:

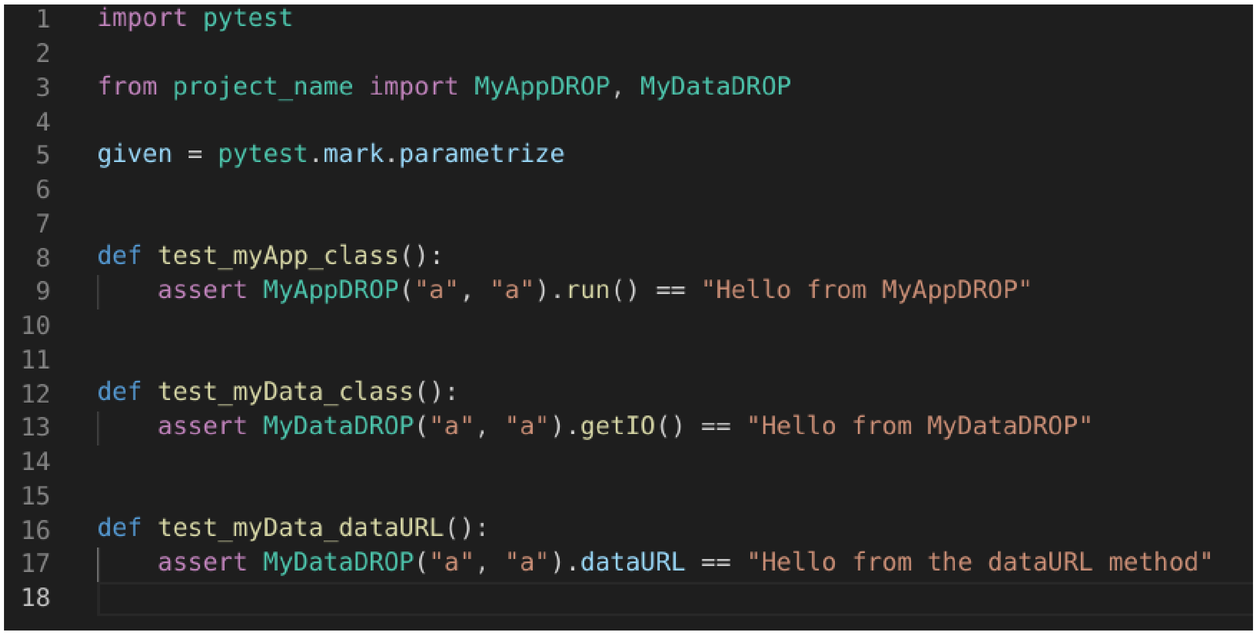
and replace the string `MyAppDROP` with `MyFirstAppDROP` everywhere. Save the file and execute the test again.:

Alright, that looks pretty serious (as expected)! It actually states that it failed in the file `__init__.py`, thus let's fix this by replacing `MyAppDROP` with `MyFirstAppDROP` there and run pytest again:

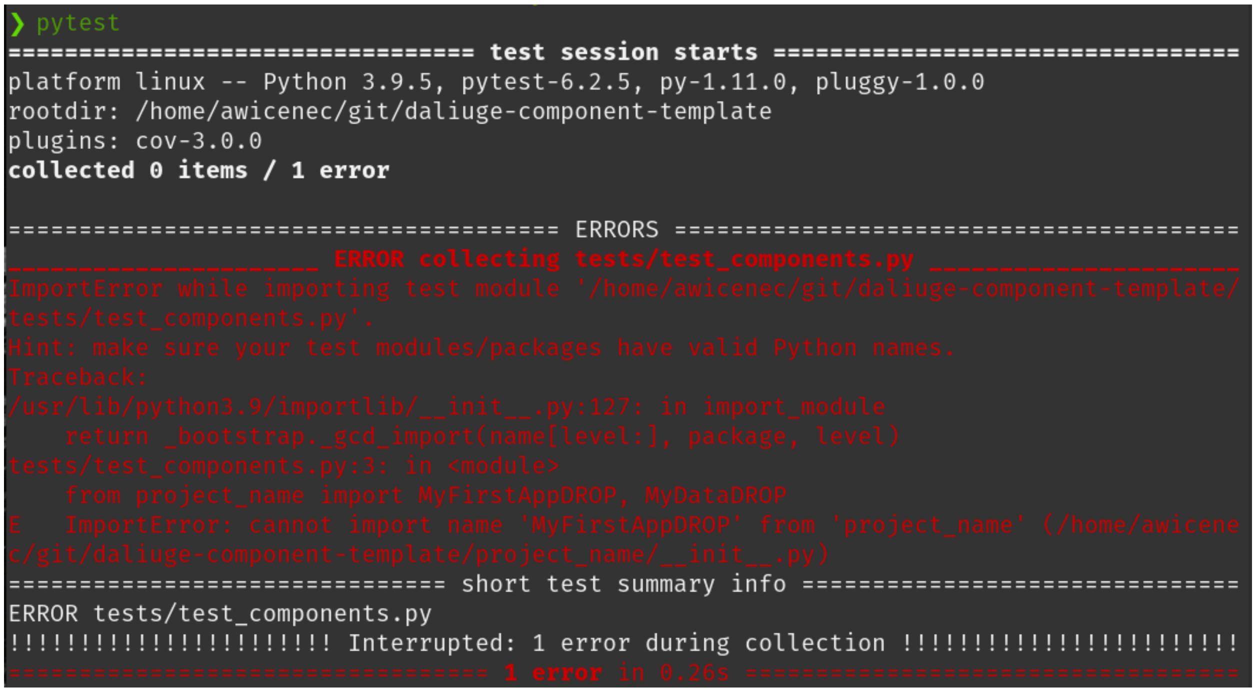
Oops, that still fails! This time in the actual `appComponents.py` file. Let's do the same replace in that file and run pytest again:

GREAT! In exactly the same manner you can work along to change the functionality of your component and always keep the tests up-to-date.

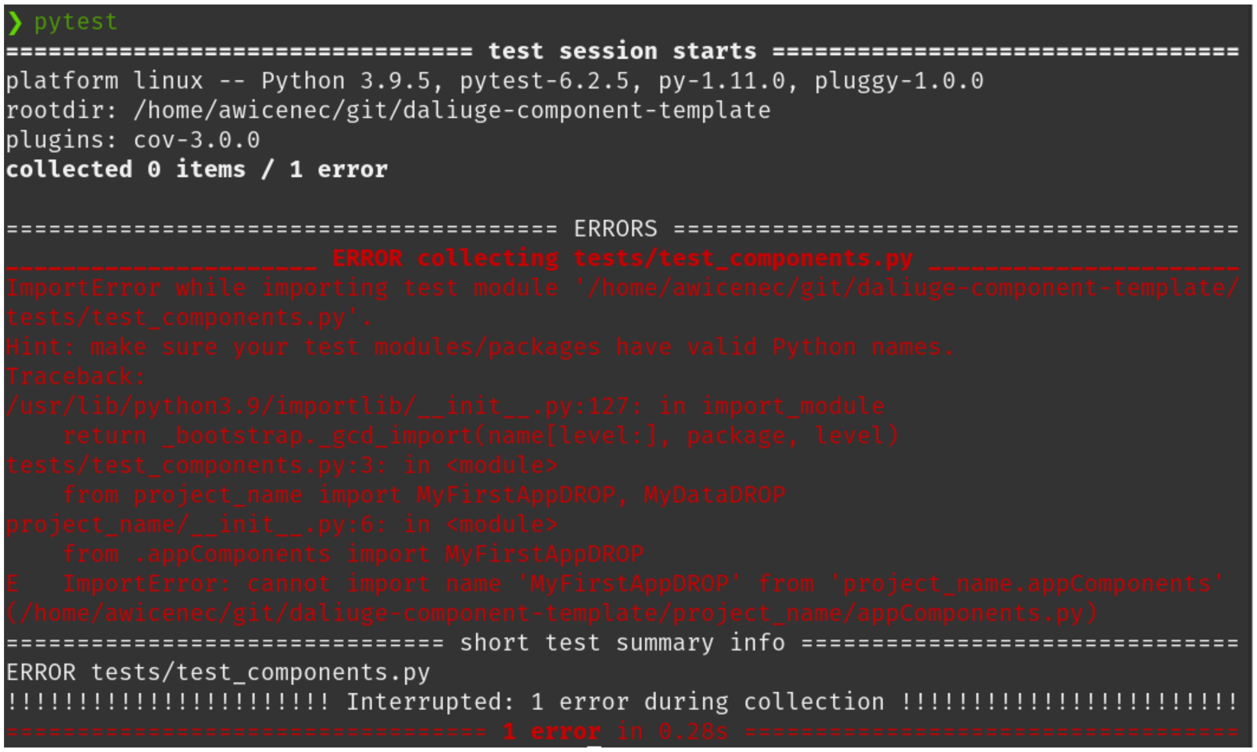
Obviously you can add more than one component class to the file `app_components.py`, or add multiple files to the directory. Just don't forget to update the file `__init__.py` accordingly as well.



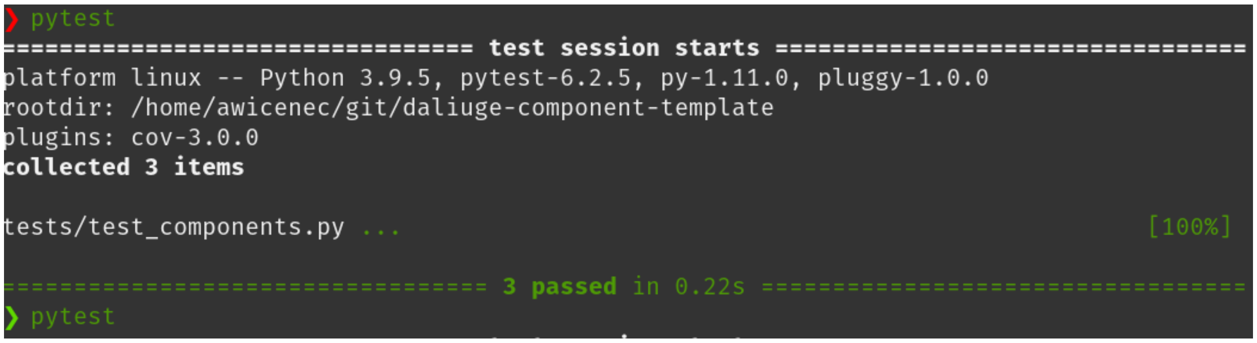
```python
import pytest

from project_name import MyAppDROP, MyDataDROP

given = pytest.mark.parametrize


def test_myApp_class():
    assert MyAppDROP("a", "a").run() == "Hello from MyAppDROP"


def test_myData_class():
    assert MyDataDROP("a", "a").getIO() == "Hello from MyDataDROP"


def test_myData_dataURL():
    assert MyDataDROP("a", "a").dataURL == "Hello from the dataURL method"

```



```
❯ pytest
================================= test session starts ==================================
platform linux -- Python 3.9.5, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/awicenec/git/daliuge-component-template
plugins: cov-3.0.0
collected 0 items / 1 error

======================================== ERRORS ========================================
_____________________ ERROR collecting tests/test_components.py ______________________
ImportError while importing test module '/home/awicenec/git/daliuge-component-template/
tests/test_components.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/usr/lib/python3.9/importlib/__init__.py:127: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests/test_components.py:3: in <module>
    from project_name import MyFirstAppDROP, MyDataDROP
E   ImportError: cannot import name 'MyFirstAppDROP' from 'project_name' (/home/awicene
c/git/daliuge-component-template/project_name/__init__.py)
=============================== short test summary info ================================
ERROR tests/test_components.py
!!!!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!!!!!
=================================== 1 error in 0.26s ===================================
```



```
❯ pytest
================================== test session starts ===================================
platform linux -- Python 3.9.5, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/awicenec/git/daliuge-component-template
plugins: cov-3.0.0
collected 0 items / 1 error

======================================== ERRORS ==========================================
______________________ ERROR collecting tests/test_components.py ______________________
ImportError while importing test module '/home/awicenec/git/daliuge-component-template/
tests/test_components.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
/usr/lib/python3.9/importlib/__init__.py:127: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests/test_components.py:3: in <module>
    from project_name import MyFirstAppDROP, MyDataDROP
project_name/__init__.py:6: in <module>
    from .appComponents import MyFirstAppDROP
E   ImportError: cannot import name 'MyFirstAppDROP' from 'project_name.appComponents'
(/home/awicenec/git/daliuge-component-template/project_name/appComponents.py)
================================ short test summary info =================================
ERROR tests/test_components.py
!!!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!!!!
=================================== 1 error in 0.28s ===================================
```



```
❯ pytest
================================== test session starts ===================================
platform linux -- Python 3.9.5, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/awicenec/git/daliuge-component-template
plugins: cov-3.0.0
collected 3 items

tests/test_components.py ...                                                      [100%]

=================================== 3 passed in 0.22s ===================================
❯ pytest
```

Remove boilerplate and add your documentation

Next step is to clean up the mess from the boylerplate template and update the documentation of our new DALiuGE component. The first thing is to remove the files *ABOUT_THIS_TEMPLATE.md* and *CONTRIBUTING.md*. The next step is to update the file *README.md*. Open that file and remove everything above `<!-- DELETE THE LINES ABOVE THIS AND WRITE YOUR PROJECT README BELOW -->` and then do exactly what is written on that line: *Write your project README below!*. Then save the file. Make sure the LICENSE file contains a license you (and your employer) are happy with. If you had to install any additional Python packages, make sure they are listed in the `requriements-test.txt` and `requirements.txt` files and modify the file `setup.py` as required. Finally add more detailed documentation to the docs directory. This will then also be published on readthedocs whenever you push to the main branch. After that you will have a pretty nice and presentable component package already.

Adding Parameters and App Arguments

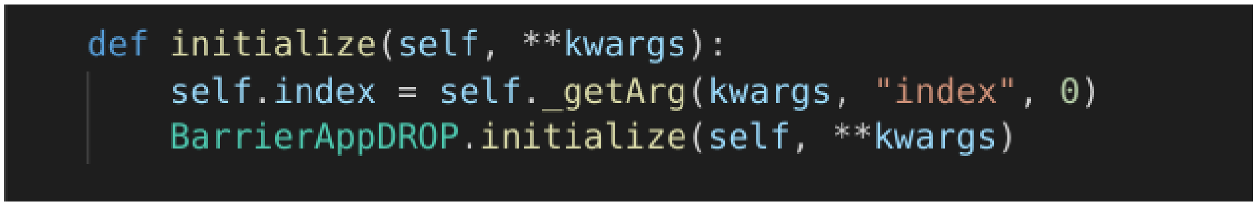
Typically workflows require some user configuration in addition to data. DALiuGE supports this in the form of parameters and/or app arguments and the end user of your component will be able to populate the values for such components in EAGLE during the development of the workflows. In order to make this happen you will need to declare the parameters through the component interface and also document them appropriately so that EAGLE can provide the parameters in the component palette to the end user. Since the end-users of your component will want to specify the values of these parameters through the EAGLE editor there are a few tricks required to enable that. For you as the developer of a component this is pretty much invisible, but you need to use the API. DALiuGE is currently offering six types of parameters:

1. dlg_string_param
2. dlg_int_param
3. dlg_float_param
4. dlg_bool_param
5. dlg_enum_param
6. dlg_list_param
7. dlg_dict_param

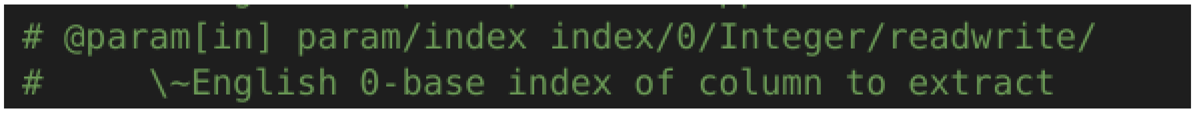
For example to define a greeting parameter for a HelloWorld application you can use a line like

```
greet = dlg_int_param("index", 0)
```

as a member of the custom component class. At runtime the param will be passed on through the graph to the component and converted to the string type after class initialization. Another example is shown below, if you have a parameter called `index` you can get the value from the graph at run time by adding a single line to your `initialize` method:


```
def initialize(self, **kwargs):
    self.index = self._getArg(kwargs, "index", 0)
    BarrierAppDROP.initialize(self, **kwargs)
```

you should always do that before calling the initialize of the base class, in the example the `BarrierAppDROP` class and add an appropriate variable to the object (`self.index`) such that all other methods will have access to the index parameter's value. Then you should also add a line to the doxygen in-line documentation like this:



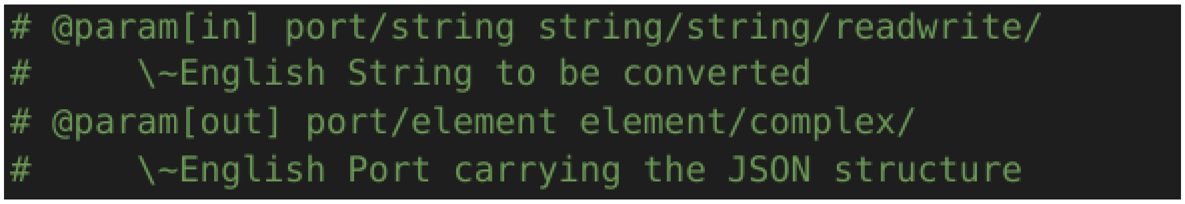
```
# @param[in] param/index index/0/Integer/readwrite/
#     \~English 0-base index of column to extract
```

see chapter *Component Description Generation* for more details on the syntax. When you now checkin your code to the github repo a github action will generate the palette (JSON description of your components) automatically and you can load it into EAGLE to construct a workflow.

Adding Input and Output Ports

Ports are how runtime data and information move in and out of your component. Ports are always connected to data components and provide the application component with a homogeneous I/O interface. App components can write whatever data you want to an output port, but be aware that other components, maybe not developed by yourself, will need a compatible reader to interpret the data. In the same spirit you might not be responsible for what is presented to your component on the input ports, but you certainly need to be able to read and use that information. See chapter *DataDROP I/O* for more details.

The first step to make sure this will fit in a workflow, is to document your own inputs and outputs and check the data on the inputs for compliance with what you are expecting. DALiuGE, or more precisely EAGLE is using that information to guide the users developing a workflow and by default allows connections only between matching ports. Again this is based on the doxygen description of your components ports, which look like this:



```
# @param[in] port/string string/string/readwrite/
#     \~English String to be converted
# @param[out] port/element element/complex/
#     \~English Port carrying the JSON structure
```

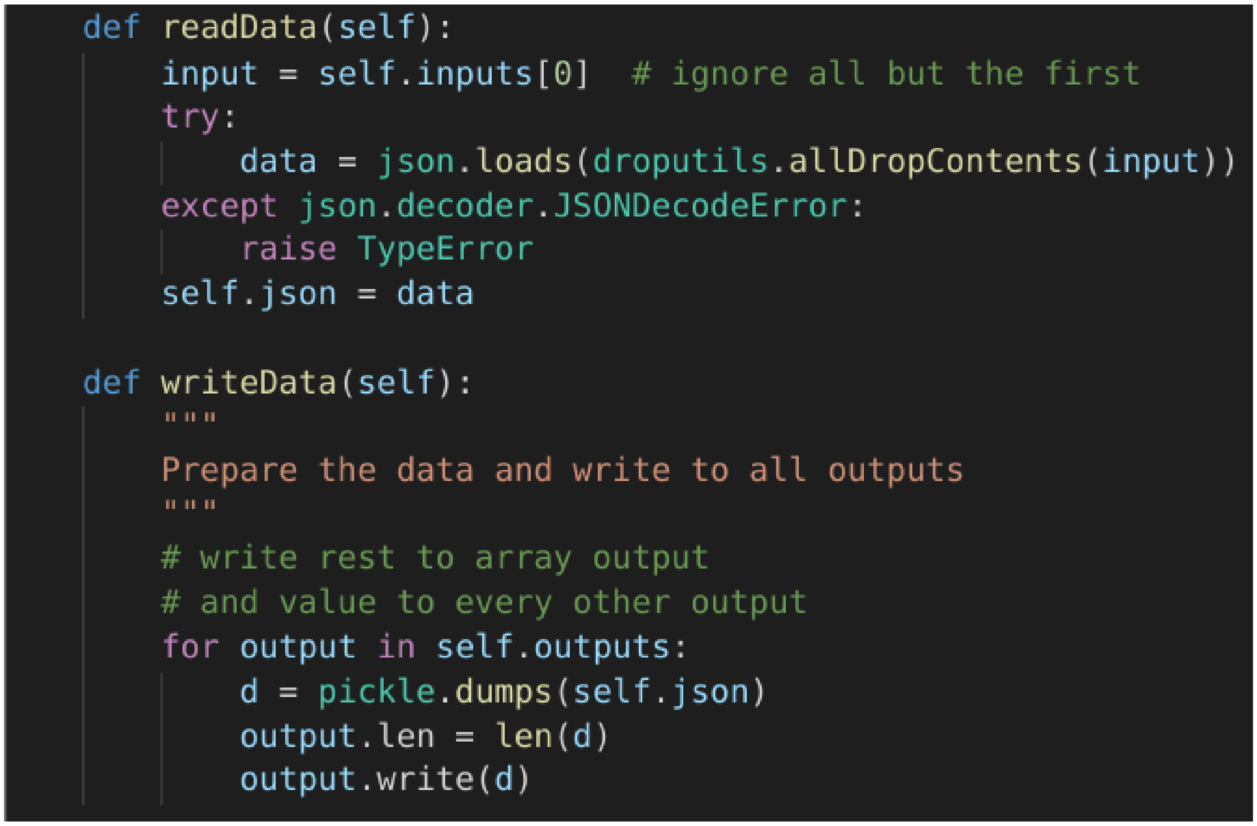
again the details for the syntax are described in the chapter *Component Description Generation*. Acessing and using the ports in your component follows always the same pattern and it might be good to separate the reading and writing part out into explicit class methods, although that is not stricly required:

In the example above the component is expecting some JSON compatible string on a single input port and it will write some JSON in a pickled format to all of its outputs. It is not required to use pickle, but it helps in a distributed environment. The input port does expect a plain string, not a pickled string in this particular case.

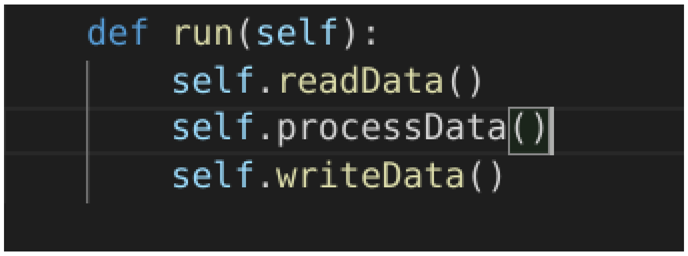
Your `run` method could look very simple and essentially always the same, but that depends on the details and complexity of the component itself. Remember that the `run` method is the only required method in a component and the only one actually called during run-time directly. The DALiuGE engine is instantiating the component and calls run, when it is triggered.

Consider Granularity and Parallelism

You can put very complex and even complete applications inside a component, but this limits code reusability and daliuge only provides scheduling and deployment parallelism down to the component level. In fact components should perform quite limited tasks, which should in general be useful for other, ideally many workflows. There is always a trade-off between overhead and functionality as well. Although the template makes the development of components quite easy, it still is an overhead, compared to just adding a few lines of code in some existing component. One of the driving requirements to write a new component might thus be whether the functionality of the new component is generic enough to be useful. There might also be other ways of implementing that same functionality and thus there might be a choice of components providing that.



```
    def readData(self):
        input = self.inputs[0]  # ignore all but the first
        try:
            data = json.loads(droputils.allDropContents(input))
        except json.decoder.JSONDecodeError:
            raise TypeError
        self.json = data

    def writeData(self):
        """
        Prepare the data and write to all outputs
        """
        # write rest to array output
        # and value to every other output
        for output in self.outputs:
            d = pickle.dumps(self.json)
            output.len = len(d)
            output.write(d)
```



```
    def run(self):
        self.readData()
        self.processData()
        self.writeData()
```

The other, really important consideration is parallelism. In general you should never do that inside a component, but leave that to the developer of the workflow itself. DALiuGE is mainly about distributing and optimizing the distribution of such parallel tasks (instances of components). You should aim to give the DALiuGE engine as many degrees of freedom as possible to deploy the final workflow on the available platform. When developing a component you won't know in what kind of workflows it is going to be used, nor will you know how big and complex those workflows are. Thus, don't assume anything and implement just the functionality to deal with a single, atomic entity of the data the component has to deal with. That also makes the implementation easier and much more straight forward.

8.3.3 Specialized Components

In addition users can develop a number of specialized components, which are based on dedicated base classes.

1. Start and Stop Components
2. Branch Components
3. MPI Components
4. Archiving/store Components

Descriptions TODO

8.3.4 Built-in Application Components

DALiuGE comes with a set of built-in components for testing and further use:

1. RPC Component
2. SCP Component

8.3.5 Dynlib Components

TODO

8.3.6 Docker Components

TODO

8.3.7 Service Components

TODO

8.3.8 Pyfunc Components

Pyfunc components are generalized python components that can be configured to behave as a custom python component entirely through component parameters and application arguments. A pyfunc component maps directly to an existing python function or a lambda expression, named application arguments and input ports are mapped to the function keyword args, and the result is mapping to the output port.

Port Parsers

Pyfunc components when interfacing with data drops may utilize one of several builtin port parsing formats.

- Pickle - Reads and writes data to pickle format
- Eval - Reads data using eval() function and writes using repr() function
- Npy - Reads and writes to .npy format
- Path - Reads the drop path rather than data
- Url - Reads the drop url rather than data

Note

Only a single port parser can currently be used for all input ports of a Pyfunc. This is subject to change in future.

8.3.9 DataDROP I/O

An application's input and output drops are accessed through its `inputs` and `outputs` members. Both of these are lists of `drops`, and will be sorted in the same order in which inputs and outputs were defined in the Logical Graph. Each element can also be queried for its `uid`.

Data can be read from input drops, and written in output drops. To read data from an input drop, one calls first the drop's `open` method, which returns a descriptor to the opened drop. Using this descriptor one can perform successive calls to `read`, which will return the data stored in the drop. Finally, the drop's `close` method should be called to ensure that all internal resources are freed.

Writing data into an output drop is similar but simpler. Application authors need only call one or more times the `write` method with the data that needs to be written.

Serialization

Many data components are capable of storing data in multiple formats determined by the drop component. The common data io interface allows app components to be compatible with many data component types, however different app components connected to the same data component must use compatible serialization and deserialization types and utilities.

String Serialization

Raw String

The simplest deserialization format supported directly by *DataDrop.write* and *DataDrop.read*.

JSON (.json)

Portable javascript object format encoded in utf-8. JSON Schema is to be handled by the input and output apps, which may also be stored as JSON. Serialization of python dictionaries is provided by *json.dump* and deserialization with *json.load*.

INI (.ini)

Simple format for storing string key-value pairs organized by sections that is supported by the python *configparser* library. Due to the exclusive use of string types this format is a good for mapping directly to command line arguments.

YAML (.yaml)

Markup format with similar featureset to JSON but additionally contains features such as comments, anchors and aliases which make it more human friendly to write. Serialization of dictionaries is provided by *yaml.dump* and deserialization with *yaml.load*.

XML (.xml)

Markup format with similar features to YAML but with the addition of attributes. Serialization can be performed using *dicttoxml* or both serialization and deserialization using *xml.etree.ElementTree*.

Python Eval (.py)

Python expressions and literals are valid string serialization formats whereby the string data is iterpreted as python code. Serialization is typically performed using the *__repr__* instance method and deserialization using *eval* or *ast.eval_literal*.

Binary Serialization

Data drops may also store binary formats that are typically more efficient than string formats and may utilize the python buffer protocol.

Raw Bytes

Data drops can always be read as raw bytes using *droputils.allDropContents* and written to using *DataDROP.write*. Reading as a bytes object creates a readonly in-memory data copy that may not be as performant as other drop utilities.

Pickle (.pkl)

Default serialazation format capable of serializing any python object. Use *save_pickle* for serialization to this format and *load_pickle* for deserialization.

Numpy (.npy)

Portable numpy serialization format. Use *save_numpy* for serialization and *load_numpy* for deserialization.

Numpy Zipped (.npz)

Portable zipped numpy serialization format. Consists of a .zip directory holding one or more .npy files.

Table Serialization

parquet (.parquet)

Open source column-based relational data format from Apache.

Specialized Serialization

Data drops such as RDBMSDrop drops manage their own record format and are interfaced using relational data objects such *dict*, *pyarrow.RecordBatch* or *pandas.DataFrame*.

8.3.10 Wrap Existing Code

TODO

8.3.11 Test And Debug

TODO

8.3.12 Component Description Generation

In order to present graph developers with well defined components for their workflow development, EAGLE uses descriptions of the components based on a JSON schema. Typically a number of these component descriptions are saved and used together in a so-called *palette*. The DALiuGE system provides two ways to create such palettes. One internal to EAGLE and another one by using special Doxygen markup inline with the component code. The latter method allows the component developer to keep everything required to describe a component in a single place, together with the code itself. The manual one allows graph developers to define and use components, which are otherwise not available, like for example bash components.

Automatic EAGLE Palette Generation

The automatic generation of a *palette* involves three steps:

1. Markup of code using custom Doxygen comments
2. Running of xml2palette.py, which is a small python script that uses the Doxygen documentation comments to generate a EAGLE palette with the required JSON format.
3. (optional) commit the resulting palette file to a graph repository.

The last two steps can be integrated into a CI build system and would then be executed automatically with any commit of the component source code. Very often one directory of source code contains multiple source files, each of which contain multiple components. The resulting palette will include descriptions of all the components found in a directory.

Generate palette using xml2palette.py

The xml2palette.py script is located in the tools directory within the DALiuGE repository. It is designed to generate a single palette file for a input directory containing doscumented code. The script has the following dependencies:

1. Doxygen
2. xsltproc

The xml2palette.py script can be run using this command line:

```
python3 xml2palette.py [-h] [-m MODULE] [-t TAG] [-c] [-r] [-s] [-v] idir ofile

positional arguments:
  idir                   input directory path or file name
  ofile                  output file name

optional arguments:
  -h, --help             show this help message and exit
  -m MODULE, --module MODULE
                         Module load path name
  -t TAG, --tag TAG      filter components with matching tag
  -c                     C mode, if not set Python will be used
  -r, --recursive        Traverse sub-directories
  -s, --parse_all        Try to parse non DALiuGE compliant functions and methods
  -v, --verbose          increase output verbosity

If no tag is specified, all components found in the input directory will part of the␣
↪output file. If, however, a tag is specified, then only those components with a␣
↪matching tag will be part of the output. Tags can be added to the Doxygen comments for␣
↪a component using:
```

```
# @param tag <tag_name>
```

Component Doxygen Markup Guide

In order to support the direct usage of newly written application components in the EAGLE editor, the DALiuGE system supports a custom set of Doxygen directives and tools. When writing an application component, developers can add specific custom Doxygen comments to the source code. These comments describe the component and can be used to automatically generate a JSON DALiuGE component description which in turn can be used in the *EAGLE*. A few basic rules to remember:

1. The DALiuGE specific comments should be contained within a *EAGLE_START* and *EAGLE_END* pair.
2. The *category* param should be set to *DynlibApp* for C/C++ code, and *PythonApp* for Python code.
3. The *construct* param should be set to Scatter or Gather. Or omitted entirely for components that will not be embedded inside a construct.

The additional comments describe both the input/output ports for a component, and the parameters of a component. Shown below are example comments for C/C++ and Python applications.

Construct

If a component is intended to implement a scatter or gather construct, then the *construct* param should be added to the Doxygen markup. When a component is flagged with a construct param, the component will be added to the palette as usual, but the component will also be added to the palette a second time embedded within an appropriate construct. Here is an example usage:

```
# @param construct Scatter
```

Parameters

Component Parameters are specified using the "param" command from doxygen. The command is followed by the name of the parameter, followed by a description. We encode multiple pieces of information within the name and description. The name must begin with "param/". This is used to disambiguate from ports, described later. The "param/" prefix will be removed during processing and only the remainder of the name will appear in the component. Names may not contain spaces. The description contains five pieces of information, separated by '/' characters: a user-facing name, the default value, the parameter type, an access descriptor (readonly/readwrite), and a "precious" flag. Note that the first line of the description must end with a '/' character.

```
# @param <internal_name> <user-facing name>/<default_value>/<type>/<field_type>/<access_
↪descriptor>/<options>/<precious>/<positional>/<description>
#
# e.g.
#
# @param start_frequency Start Frequency/500/Integer/ComponentParameter/readwrite//False/
↪False/
#     \~English the start frequency to read from
#     \~Chinese
```

The **precious** flag indicates that the value of the parameter should always be shown to the user, even when the parameter contains its default value. The flag also enforces that the parameter will always end-up on the command line, regardless of whether it contains the default value.

The **positional** flag indicates that this parameter is a positional argument on a command line, and will be added to the command line without a prefix.

Component Parameters vs. Application Arguments

There are two different types of parameter that can be specified on a component. These two types are: Component Parameter and Application Argument. Component parameters are intended to direct the behaviour of the DALiuGE component itself, while Application arguments are intended to direct the application underneath the component. For example, a component may have Component Parameter describing the number of CPUs to be used for execution, but a application argument for the arguments on the command line for the component.

The two types of parameters use different keywords (ComponentParameter vs. ApplicationArgument), as shown in the example below.

```
# @param start_frequency Start Frequency/500/Integer/ComponentParameter/readwrite//False/
↪False/
#     \~English the start frequency to read from
* @param method Method/mean/Select/ApplicationArgument/readwrite/mean,median/False/False/
*     \~English The method used for averaging
```

Parameter Types

Available types are:

1. String
2. Integer
3. Float
4. Boolean
5. Select
6. Password
7. Json
8. Python
9. Object

The Select parameters describe parameters that only have a small number of valid values. The valid values are specified in the “options” part of the Doxygen command, using a comma separated list. For example:

```
* @param method Method/mean/Select/ApplicationArgument/readwrite/mean,median/False/False/
*     \~English The method used for averaging
```

All other parameter types have empty options.

Ports

Component ports are (somewhat confusingly) also specified using the “param” from doxygen. However, field types of InputPort and OutputPort are used.

```
# @param <internal_name> <user-facing name>/<default_value>/<type>/<field_type>/<access_
↪descriptor>/<options>/<precious>/<positional>/<description>
#
# e.g.
```

(continues on next page)

(continued from previous page)

```
#
# @param config Config//String/InputPort/readwrite//False/False/
#     \~English the configuration of the input_port
#     \~Chinese
```

Complete example for C/C++

```
/*!
* \brief Load a CASA Measurement Set in the DaliugeApplication Framework
* \details We will build on the LoadParset structure - but use the contents
* of the parset to load a measurement set.
* \par EAGLE_START
* \param category DynlibApp
* \param start_frequency Start Frequency/500/Integer/ComponentParameter/readwrite//False/
↪False/
*     \~English the start frequency to read from
*     \~Chinese
* \param end_frequency End Frequency/500/Integer/ComponentParameter/readwrite//False/
↪False/
*     \~English the end frequency to read from
*     \~Chinese
* \param channels Channels/64/Integer/ApplicationArgument/readonly//False/False/
*     \~English how many channels to load
*     \~Chinese
* \param method Method/mean/Select/ApplicationArgument/readwrite/mean,median/False/False/
*     \~English The method used for averaging
* \param config Config//String/InputPort/readwrite//False/False/
*     \~English the configuration of the input_port
*     \~Chinese
* \param event Event//Event/InputPort/readwrite//False/False/
*     \~English the event of the input_port
*     \~Chinese
* \param file File//File/OutputPort/readwrite//False/False/
*     \~English the file of the output_port
*     \~Chinese
* \par EAGLE_END
*/
```

Complete example for Python

```
##
# @brief Load a CASA Measurement Set in the DaliugeApplication Framework
# @details We will build on the LoadParset structure - but use the contents
# of the parset to load a measurement set.
# @par EAGLE_START
# @param category PythonApp
# @param start_frequency Start Frequency/500/Integer/ComponentParameter/readwrite//False/
↪False/
```

(continues on next page)

(continued from previous page)

```
#     \~English the start frequency to read from
#     \~Chinese
# @param end_frequency End Frequency/500/Integer/ComponentParameter/readwrite//False/
↪False/
#     \~English the end frequency to read from
#     \~Chinese
# @param channels Channels/64/Integer/ApplicationArgument/readonly//False/False/
#     \~English how many channels to load
#     \~Chinese
# @param method Method/mean/Select/ApplicationArgument/readwrite/mean,median/False/False/
#     \~English The method used for averaging
# @param config Config//String/InputPort/readwrite//False/False/
#     \~English the configuration of the input_port
#     \~Chinese
# @param event Event//Event/InputPort/readwrite//False/False/
#     \~English the event of the input_port
#     \~Chinese
# @param file File//File/OutputPort/readwrite//False/False/
#     \~English the file of the output_port
#     \~Chinese
# @par EAGLE_END
```

Manual EAGLE Palette Generation

The *palette* and *logical graph* JSON formats are almost interchangable. The two formats differ only by filename extension and by a single attribute in the JSON contents (modelData.fileType is "graph" versus "palette"). In fact one can save a graph as a palette. Defining a component in EAGLE requires the activation of the *palette mode*. More details can be found in the EAGLE documentation.

8.3.13 Deployment Testing

TODO

- genindex
- search

8.4 DALiuGE *Data* Component Developers Guide

This chapter describes what developers need to do to write a new data component that can be used as a Data Drop during the execution of a DALiuGE graph.

Different from most other frameworks DALiuGE makes data components first class entities in the context of a workflow. In fact data components, or rather the instances of data components, the Data Drops, are driving the execution of a workflow. Consequently DALiuGE graphs are showing both application and data components as graph nodes. Edges in DALiuGE graphs are symbolising event flow and not data flow. In fact most Data Drops just refer to their data payload using a URL. The Data Drop layer provides two main features:

- Abstraction of the I/O interface particularities of the underlying data storage mechanism.
- Implementation of the DALiuGE data state engine, which is the DALiuGE mechansim to drive the execution of the workflow graphs.

8.4.1 Components

Filesystem Components

Path Based Drop

`PathBasedDrop` is an interface for retreiving the path for drops that are backed by a filesystem such as local filesystem, NFS or MPFS. Many libraries either have or only have support for reading and writing with a filesystem path.

File Drop

`FileDROP` is a highly compatible data drop type that can be easily used as persistent volume I/O and inspection of individual app component I/O. The downside of using file drops is reduced I/O performance compared to alternative memory based drops that can instead utilize buffer protocol.

Environment variables can be used in the the file path location using the '$' literal, e.g. '$DLG_ROOT' evaluates to '/home/username/dlg'

Container Drop (Legacy)

`ContainerDROP`

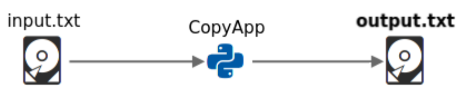
Directory Container Drop (Legacy)

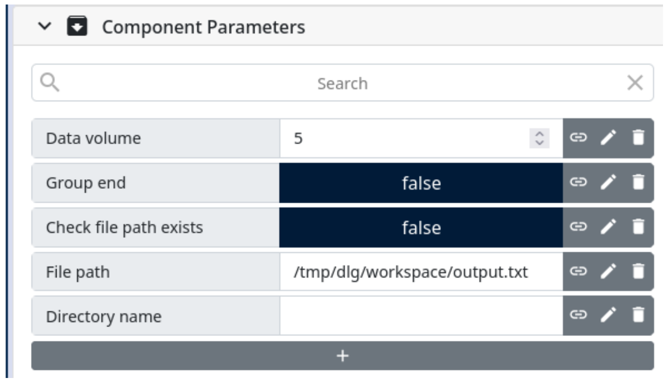
`DirectoryContainer`

Using Filesystem Components as Persistent Volume Storage

Filesystem component paths are relative to the temporary DALiuGE workspace but may also be of absolute path to the the machine running DALiuGE engine of connected app drops, most commonly the default DALiuGE shared filesystem mount location '/tmp/dlg/workspace` located on both the host machine and daliuge engine virtual machine.

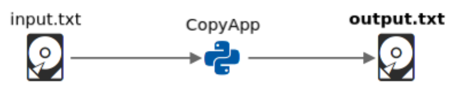
In the following example graph, both the input.txt file drop and output.txt file drop paths point to a persistent absolute location in the workspace folder. On successful graph execution, the output location will be updated.

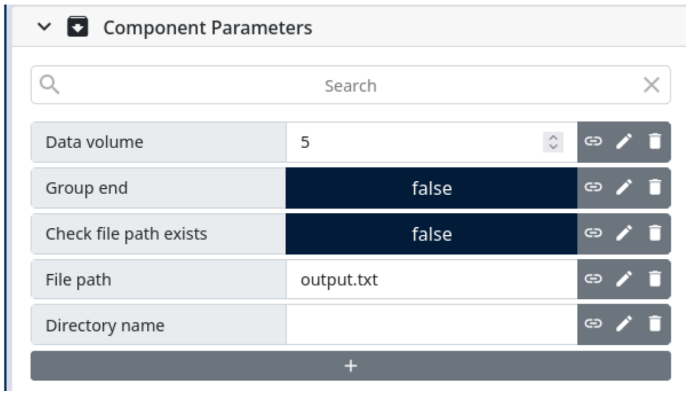




Data volume	5
Group end	false
Check file path exists	false
File path	/tmp/dlg/workspace/output.txt
Directory name	

Depending on DALiuGE settings, the relative workflow path will also be generated and made persistent. Setting the output path to a relative location will populate a new workflow directory with output after every successful execution.





Data volume	5
Group end	false
Check file path exists	false
File path	output.txt
Directory name	

Memory Components

In Memory Drop

`InMemoryDROP` Is the simplest and most efficient data drop where data is stored in a python binary buffer object. In memory drops are good for single threaded execution patterns where data persistences is not needed before and after the workflow executes.

Shared Memory Drop

`SharedMemoryDROP` Is a special in-memory drop that allows multiple processes to read from the same memory location. See *Shared Memory* for further information.

Buffer Protocol

Python buffer protocol is a special interface available for numerous python objects synonymous to byte array syntax in C language, whereby python objects expose their raw bytes to other python objects. Memory components exposed via the buffer protocol drop interface can benefit from zero-copy reading and slicing by app components.

To implement `DataIO.buffer` it is recommended to ensure zero copying is performed with memory based drops for greatly improved drop read performance.

Note: String types in python3 use utf-8 character slicing and hence are not compatible with buffer protocol due to characters not being of fixed size. Consider supporting encoded strings in app drop I/O.

Plasma Components

Plasma Drops are a special shared memory drop where memory is managed by an external data store process that is selected using a filesystem socket. The Plasma data store counts Plasma object references which are automatically incremented decremented by DALiuGE.

Plasma allows direct memory access between DALiuGE, native apps and docker apps that link to the apache arrow Plasma library.

Plasma Drop

`PlasmaDROP` is the basic Plasma component implementation that serializes data to a fixed size Plasma buffer allocated in a Plasma store. A Plasma socket filesystem location must be specified. Daliuge will automatically populate the drop PlasmaID parameter.

It is worth noting that PlasmaDrop can only share memory to processes and virtual machines running on the same physical machine. In a compute cluster stream proxies will be created to transfer data across nodes.

PlasmaFlight Drop (Experimental)

`PlasmaFlightDROP` is an extended Plasma Drop implementation using Apache Arrow Flight for network communication (details at https://github.com/ICRAR/plasmaflight).

Arrow Flight queries can be made by using a Plasma ObjectID as the flight ticket.

PlasmaFlight drops are effective for sharing

PlasmaFlight Service

A service application that can host both the Plasma DataStore and PlasmaFlight server for Plasma components.

NGAS Components

NgasDROP

`NgasDROP` A data drop that stores data on the localfilsystem then moves the data buffer to Next Generation Archive System (NGAS). NGAS only supports writing new files and not appending to an existing FileId. See *https://ngas.readthedocs.io* for more information on NGAS.

NOTE: The Daliuge Component Developement Guide is work in progress!

RDBMS Components

RDBMS Drop (Legacy)

`RDBMSDrop` is a generic relational database storage drop that uses SQL to insert and select data. Instead of using the getIO() interface the methods *insert* and *select* are available.

NOTE: The Daliuge Component Developement Guide is work in progress!

Cloud Storage Components

AWS S3 Drop (Experimental)

`S3DROP` Uses AWS S3 storage for containing drop contents using the AWS SDK package boto3.

NOTE: The Daliuge Component Developement Guide is work in progress!

Parameter Components

ParameterSet Drop

`ParameterSetDROP` Defines a set of app parameters in EAGLE and stores them in a in memory drop. Supported modes include: YANDA(INI)

8.4.2 Custom Components

Writing Data Components

Data I/O

A data components' input and output methods are defined by the abstract class `DataIO`. The methods in that class are just empty definitions and have to be implemented by the actual data component.

NOTE: The Daliuge Component Developement Guide is work in progress!

Graph Translation

DALiuGE uses `dlg.common.Categories`, `dlg.common.STORAGE_TYPES` and `dlg.graph_loader.STORAGE_TYPES` to serialize and deserialize logical graph data drops. These must be extended for new data drop types to be processed by the translator.

NOTE: The Daliuge Component Developement Guide is work in progress!

Eagle Custom Data Drops Integeration

NOTE: The Daliuge Component Developement Guide is work in progress!

Should you have any questions, please contact us at: dfms_prototype AT googlegroups DOT com

8.5 Citations

As you use DALiuGE for your exciting projects, please cite the following paper:

Wu, C., Tobar, R., Vinsen, K., Wicenec, A., Pallot, D., Lao, B., Wang, R., An, T., Boulton, M., Cooper, I. and Dodson, R., 2017. DALiuGE: A Graph Execution Framework for Harnessing the Astronomical Data Deluge. Astronomy and Computing, 20, pp.1-15. (2017)

CHAPTER

NINE

DALIUGE USERS GUIDE

9.1 Users Guide Introduction

The DALiuGE system can and has been used on a huge range of different size systems. It can easily be deployed on a person's laptop, across multiple personal machines as well as small and large compute clusters. That flexibility comes with some difficulty in describing how the system is intended to be used, since that is obviously dependent on the way it is deployed. This guide mainly describes the basic usage of the system as it would appear when deployed on a single machine or a small cluster. Other deployment scenarios and their respective differences are described in the *Operational concepts* chapter. The purpose of DALiuGE is to allow users to develop and execute complex parallel workflows and as such it's real strength only shines when it comes to massive deployments. However, the basic usage does not really change at all and many real-life, mostly weak scaling workflows can be scaled up and down by changing just one or a few parameters.

Hopefully you will be able to identify yourself with one (or more) of the five user groups:

1. Scientists who want to reduce their data using a existing workflows.
2. Scientists/developers who want to design a new workflow using existing components.
3. Scientists/developers who want to develop new component descriptions.
4. Developers who want to develop new components.
5. Developers who want to develop a new algorithm.

This guide covers only the frist two groups, the last three are covered in the *DALiuGE Component Developers Guide* chapter. There is also a *Startup and Shutdown Guide* section to explain how to use the system in small deployments.

This guide does also not cover the usage of the EAGLE editor in any more detail than required, since that is covered in the EAGLE documentation.

9.2 Data Reduction User's Guide

TODO

9.3 Graph Developer's Guide

TODO

CHAPTER

TEN

CLI USER GUIDE

As briefly highlighted in the *Startup and Shutdown Guide* there is a complete Command Line Interface (CLI) available to control the managers and translate, partition and deploy graphs. This makes the whole system independent of EAGLE or a web browser and also allows the system to be scripted (although we recommend to do this in Python following the *API Documentation*). The available functionality of the CLI depends on which parts of the DALiuGE execution framework are actually installed on the python virtualenv.

10.1 Basic Usage

In order to be able to use the CLI at least daliuge-common needs to be installed. In that case the functionality is obviously very limited, but it shows already the basic usage:

```
 dlg
Usage: /home/awicenec/.pyenv/versions/dlg/bin/dlg [command] [options]

Commands are:
    version                     Reports the DALiuGE version and exits

Try $PATH/bin/dlg [command] --help for more details
```

If daliuge-engine is also installed it is a bit more interesting:

```
 dlg
Usage: dlg [command] [options]

Commands are:
    daemon                      Starts a DALiuGE Daemon process
    dim                         Starts a Drop Island Manager
    include_dir                 Print the directory where C header files can be found
    mm                          Starts a Master Manager
    monitor                     A proxy to be used in conjunction with the dlg proxy in␣
↪restricted environments
    nm                          Starts a Node Manager
    proxy                       A reverse proxy to be used in restricted environments to␣
↪contact the Drop Managers
    replay                      Starts a Replay Manager
    version                     Reports the DALiuGE version and exits

Try dlg [command] --help for more details
```

If *only* the daliuge-translator is installed this changes to:

```
 dlg
Usage: dlg [command] [options]

Commands are:
    fill                     Fill a Logical Graph with parameters
    lgweb                    A Web server for the Logical Graph Editor
    map                      Maps a Physical Graph Template to resources and produces a␣
↪Physical Graph
    partition                Divides a Physical Graph Template into N logical partitions
    submit                   Submits a Physical Graph to a Drop Manager
    unroll                   Unrolls a Logical Graph into a Physical Graph Template
    unroll-and-partition     unroll + partition
    version                  Reports the DALiuGE version and exits

Try dlg [command] --help for more details
```

If everything is installed the output is a merge of all three:

```
 dlg
Usage: dlg [command] [options]

Commands are:
    daemon                   Starts a DALiuGE Daemon process
    dim                      Starts a Drop Island Manager
    fill                     Fill a Logical Graph with parameters
    include_dir              Print the directory where C header files can be found
    lgweb                    A Web server for the Logical Graph Editor
    map                      Maps a Physical Graph Template to resources and produces a␣
↪Physical Graph
    mm                       Starts a Master Manager
    monitor                  A proxy to be used in conjunction with the dlg proxy in␣
↪restricted environments
    nm                       Starts a Node Manager
    partition                Divides a Physical Graph Template into N logical partitions
    proxy                    A reverse proxy to be used in restricted environments to␣
↪contact the Drop Managers
    replay                   Starts a Replay Manager
    submit                   Submits a Physical Graph to a Drop Manager
    unroll                   Unrolls a Logical Graph into a Physical Graph Template
    unroll-and-partition     unroll + partition
    version                  Reports the DALiuGE version and exits

Try dlg [command] --help for more details
```

10.2 Subcommand usage

10.2.1 Command: dlg daemon

Help output:

```
Usage: daemon [options]

Starts a DALiuGE Daemon process

Options:
  -h, --help        show this help message and exit
  -m, --master      Start this DALiuGE daemon as the master daemon
  --no-nm           Don't start a NodeDropManager by default
  --no-zeroconf     Don't enable zeroconf on this DALiuGE daemon
  -v, --verbose     Become more verbose. The more flags, the more verbose
  -q, --quiet       Be less verbose. The more flags, the quieter
```

10.2.2 Command: dlg dim

Help output:

```
Usage: dim [options]

Starts a Drop Island Manager

Options:
  -h, --help             show this help message and exit
  -H HOST, --host=HOST   The host to bind this instance on
  -P PORT, --port=PORT   The port to bind this instance on
  -m MAXREQSIZE, --max-request-size=MAXREQSIZE
                         The maximum allowed HTTP request size, in MB
  -d, --daemon           Run as daemon
  --cwd                  Short for '-w .'
  -w WORK_DIR, --work-dir=WORK_DIR
                         Working directory, defaults to '/' in daemon mode, '.'
                         in interactive mode
  -s, --stop             Stop an instance running as daemon
  --status               Checks if there is daemon process actively running
  -T TIMEOUT, --timeout=TIMEOUT
                         Timeout used when checking for the daemon process
  -v, --verbose          Become more verbose. The more flags, the more verbose
  -q, --quiet            Be less verbose. The more flags, the quieter
  -l LOGDIR, --log-dir=LOGDIR
                         The directory where the logging files will be stored
  -N NODES, --nodes=NODES
                         Comma-separated list of node names managed by this DIM
  -k PKEYPATH, --ssh-pkey-path=PKEYPATH
                         Path to the private SSH key to use when connecting to
                         the nodes
  --dmCheckTimeout=DMCHECKTIMEOUT
```

(continues on next page)

(continued from previous page)

```
                        Maximum timeout used when automatically checking for
                        DM presence
```

10.2.3 Command: dlg fill

Help output:

```
Usage: fill [options]

Fill a Logical Graph with parameters

Options:
  -h, --help            show this help message and exit
  -v, --verbose         Become more verbose. The more flags, the more verbose
  -q, --quiet           Be less verbose. The more flags, the quieter
  -o OUTPUT, --output=OUTPUT
                        Where the output should be written to (default:
                        stdout)
  -f, --format          Format JSON output (newline, 2-space indent)
  -L LOGICAL_GRAPH, --logical-graph=LOGICAL_GRAPH
                        Path to the Logical Graph (default: stdin)
  -p PARAMETER, --parameter=PARAMETER
                        Parameter specification (either 'name=value' or a JSON
                        string)
  -R, --reproducibility
                        Level of reproducibility. Default 0 (NOTHING). Accepts '-1'-'8'"
                        Refer to dlg.common.reproducibility.constants for more␣
↪explanation.
```

10.2.4 Command: dlg include_dir

Help output:

```
/home/awicenec/.pyenv/versions/3.8.10/envs/dlg/lib/python3.8/site-packages/dlg/apps
```

10.2.5 Command: dlg lgweb

Help output:

```
Usage: lgweb [options]

A Web server for the Logical Graph Editor

Options:
  -h, --help            show this help message and exit
  -d LG_PATH, --lgdir=LG_PATH
                        A path that contains at least one sub-directory, which
                        contains logical graph files
```

(continues on next page)

(continued from previous page)

```
  -t PGT_PATH, --pgtdir=PGT_PATH
                        physical graph template path (output)
  -H HOST, --host=HOST  logical graph editor host (all by default)
  -p PORT, --port=PORT  logical graph editor port (8084 by default)
  -v, --verbose         Enable more logging

If you have no Logical Graphs yet and want to see some you can grab a copy
of those maintained at:

https://github.com/ICRAR/daliuge-logical-graphs
```

10.2.6 Command: dlg map

Help output:

```
Usage: map [options]

Maps a Physical Graph Template to resources and produces a Physical Graph

Options:
  -h, --help            show this help message and exit
  -v, --verbose         Become more verbose. The more flags, the more verbose
  -q, --quiet           Be less verbose. The more flags, the quieter
  -o OUTPUT, --output=OUTPUT
                        Where the output should be written to (default:
                        stdout)
  -f, --format          Format JSON output (newline, 2-space indent)
  -H HOST, --host=HOST  The host we connect to to deploy the graph
  -p PORT, --port=PORT  The port we connect to to deploy the graph
  -P PGT_PATH, --physical-graph-template=PGT_PATH
                        Path to the Physical Graph to submit (default: stdin)
  -N NODES, --nodes=NODES
                        The nodes where the Physical Graph will be
                        distributed, comma-separated
  -i ISLANDS, --islands=ISLANDS
                        Number of islands to use during the partitioning
```

10.2.7 Command: dlg mm

Help output:

```
Usage: mm [options]

Starts a Master Manager

Options:
  -h, --help            show this help message and exit
  -H HOST, --host=HOST  The host to bind this instance on
  -P PORT, --port=PORT  The port to bind this instance on
  -m MAXREQSIZE, --max-request-size=MAXREQSIZE
```

(continues on next page)

(continued from previous page)

```
                        The maximum allowed HTTP request size, in MB
  -d, --daemon          Run as daemon
  --cwd                 Short for '-w .'
  -w WORK_DIR, --work-dir=WORK_DIR
                        Working directory, defaults to '/' in daemon mode, '.'
                        in interactive mode
  -s, --stop            Stop an instance running as daemon
  --status              Checks if there is daemon process actively running
  -T TIMEOUT, --timeout=TIMEOUT
                        Timeout used when checking for the daemon process
  -v, --verbose         Become more verbose. The more flags, the more verbose
  -q, --quiet           Be less verbose. The more flags, the quieter
  -l LOGDIR, --log-dir=LOGDIR
                        The directory where the logging files will be stored
  -N NODES, --nodes=NODES
                        Comma-separated list of node names managed by this MM
  -k PKEYPATH, --ssh-pkey-path=PKEYPATH
                        Path to the private SSH key to use when connecting to
                        the nodes
  --dmCheckTimeout=DMCHECKTIMEOUT
                        Maximum timeout used when automatically checking for
                        DM presence
```

10.2.8 Command: dlg monitor

Help output:

```
Usage: monitor [options]

A proxy to be used in conjunction with the dlg proxy in restricted
environments

Options:
  -h, --help            show this help message and exit
  -H HOST, --host=HOST  The network interface the monitor is bind
  -o MONITOR_PORT, --monitor_port=MONITOR_PORT
                        The monitor port exposed to the DALiuGE proxy
  -c CLIENT_PORT, --client_port=CLIENT_PORT
                        The proxy port exposed to the client
  -p PUBLICATION_PORT, --publication_port=PUBLICATION_PORT
                        Port used to publish the list of proxies for clients
                        to look at
  -d, --debug           Whether to log debug info
```

10.2.9 Command: dlg nm

Help output:

```
Usage: nm [options]

Starts a Node Manager

Options:
  -h, --help            show this help message and exit
  -H HOST, --host=HOST  The host to bind this instance on
  -P PORT, --port=PORT  The port to bind this instance on
  -m MAXREQSIZE, --max-request-size=MAXREQSIZE
                        The maximum allowed HTTP request size, in MB
  -d, --daemon          Run as daemon
  --cwd                 Short for '-w .'
  -w WORK_DIR, --work-dir=WORK_DIR
                        Working directory, defaults to '/' in daemon mode, '.'
                        in interactive mode
  -s, --stop            Stop an instance running as daemon
  --status              Checks if there is daemon process actively running
  -T TIMEOUT, --timeout=TIMEOUT
                        Timeout used when checking for the daemon process
  -v, --verbose         Become more verbose. The more flags, the more verbose
  -q, --quiet           Be less verbose. The more flags, the quieter
  -l LOGDIR, --log-dir=LOGDIR
                        The directory where the logging files will be stored
  -I, --no-log-ids      Do not add associated session IDs and Drop UIDs to log
                        statements
  --no-dlm              Don't start the Data Lifecycle Manager on this
                        NodeManager
  --dlg-path=DLGPATH    Path where more DALiuGE-related libraries can be found
  --error-listener=ERRORLISTENER
                        The error listener class to be used
  --event-listeners=EVENT_LISTENERS
                        A colon-separated list of event listener classes to be
                        used
  -t MAX_THREADS, --max-threads=MAX_THREADS
                        Max thread pool size used for executing drops. 0
                        (default) means no pool.
```

10.2.10 Command: dlg partition

Help output:

```
Usage: partition [options]

Divides a Physical Graph Template into N logical partitions

Options:
  -h, --help            show this help message and exit
  -v, --verbose         Become more verbose. The more flags, the more verbose
```

(continues on next page)

(continued from previous page)

```
  -q, --quiet           Be less verbose. The more flags, the quieter
  -o OUTPUT, --output=OUTPUT
                        Where the output should be written to (default:
                        stdout)
  -f, --format          Format JSON output (newline, 2-space indent)
  -N PARTITIONS, --partitions=PARTITIONS
                        Number of partitions to generate
  -i ISLANDS, --islands=ISLANDS
                        Number of islands to use during the partitioning
  -a ALGO, --algorithm=ALGO
                        algorithm used to do the partitioning
  -A ALGO_PARAMS, --algorithm-param=ALGO_PARAMS
                        Extra name=value parameters used by the algorithms
                        (algorithm-specific)
  -P PGT_PATH, --physical-graph-template=PGT_PATH
                        Path to the Physical Graph Template (default: stdin)
```

10.2.11 Command: dlg proxy

Help output:

```
Usage: proxy [options]

A reverse proxy to be used in restricted environments to contact the Drop
Managers

Options:
  -h, --help            show this help message and exit
  -d DLG_HOST, --dlg_host=DLG_HOST
                        DALiuGE Node Manager host IP (required)
  -m MONITOR_HOST, --monitor_host=MONITOR_HOST
                        Monitor host IP (required)
  -l LOG_DIR, --log_dir=LOG_DIR
                        Log directory (optional)
  -f DLG_PORT, --dlg_port=DLG_PORT
                        The port the DALiuGE Node Manager is running on
  -o MONITOR_PORT, --monitor_port=MONITOR_PORT
                        The port the DALiuGE monitor is running on
  -b, --debug           Whether to log debug info
  -i ID, --id=ID        The ID of this proxy for on the monitor side
                        (required)
```

10.2.12 Command: dlg replay

Help output:

```
Usage: replay [options]

Starts a Replay Manager

Options:
  -h, --help            show this help message and exit
  -H HOST, --host=HOST  The host to bind this instance on
  -P PORT, --port=PORT  The port to bind this instance on
  -m MAXREQSIZE, --max-request-size=MAXREQSIZE
                        The maximum allowed HTTP request size, in MB
  -d, --daemon          Run as daemon
  --cwd                 Short for '-w .'
  -w WORK_DIR, --work-dir=WORK_DIR
                        Working directory, defaults to '/' in daemon mode, '.'
                        in interactive mode
  -s, --stop            Stop an instance running as daemon
  --status              Checks if there is daemon process actively running
  -T TIMEOUT, --timeout=TIMEOUT
                        Timeout used when checking for the daemon process
  -v, --verbose         Become more verbose. The more flags, the more verbose
  -q, --quiet           Be less verbose. The more flags, the quieter
  -l LOGDIR, --log-dir=LOGDIR
                        The directory where the logging files will be stored
  -S STATUS_FILE, --status-file=STATUS_FILE
                        File containing a continuous graph status dump
  -g GRAPH_FILE, --graph-file=GRAPH_FILE
                        File containing a physical graph dump
```

10.2.13 Command: dlg submit

Help output:

```
Usage: submit [options]

Submits a Physical Graph to a Drop Manager

Options:
  -h, --help            show this help message and exit
  -v, --verbose         Become more verbose. The more flags, the more verbose
  -q, --quiet           Be less verbose. The more flags, the quieter
  -H HOST, --host=HOST  The host we connect to to deploy the graph
  -p PORT, --port=PORT  The port we connect to to deploy the graph
  -P PG_PATH, --physical-graph=PG_PATH
                        Path to the Physical Graph to submit (default: stdin)
  -s SESSION_ID, --session-id=SESSION_ID
                        Session ID (default: <pg_name>-<current-time>)
  -S, --skip-deploy     Skip the deployment step (default: False)
  -w, --wait            Wait for the graph execution to finish (default:
```

(continues on next page)

(continued from previous page)

```
                        False)
  -i POLL_INTERVAL, --poll-interval=POLL_INTERVAL
                        Polling interval used for monitoring the execution
                        (default: 10)
  -R REPRODUCIBILITY, --reproducibility=REPRODUCIBILITY
                        Fetch (and output) reproducibility data for final execution␣
↪graph.
                        (default: False)
```

10.2.14 Command: dlg unroll

Help output:

```
Usage: unroll [options]

Unrolls a Logical Graph into a Physical Graph Template

Options:
  -h, --help            show this help message and exit
  -v, --verbose         Become more verbose. The more flags, the more verbose
  -q, --quiet           Be less verbose. The more flags, the quieter
  -o OUTPUT, --output=OUTPUT
                        Where the output should be written to (default:
                        stdout)
  -f, --format          Format JSON output (newline, 2-space indent)
  -L LG_PATH, --logical-graph=LG_PATH
                        Path to the Logical Graph (default: stdin)
  -p OID_PREFIX, --oid-prefix=OID_PREFIX
                        Prefix to use for generated OIDs
  -z, --zerorun         Generate a Physical Graph Template that takes no time
                        to run
  --app=APP             Force an app to be used in the Physical Graph. 0=Don't
                        force, 1=SleepApp, 2=SleepAndCopy
```

10.2.15 Command: dlg unroll-and-partition

Help output:

```
Usage: unroll-and-partition [options]

unroll + partition

Options:
  -h, --help            show this help message and exit
  -v, --verbose         Become more verbose. The more flags, the more verbose
  -q, --quiet           Be less verbose. The more flags, the quieter
  -o OUTPUT, --output=OUTPUT
                        Where the output should be written to (default:
                        stdout)
  -f, --format          Format JSON output (newline, 2-space indent)
```

(continues on next page)

(continued from previous page)

```
  -L LG_PATH, --logical-graph=LG_PATH
                        Path to the Logical Graph (default: stdin)
  -p OID_PREFIX, --oid-prefix=OID_PREFIX
                        Prefix to use for generated OIDs
  -z, --zerorun         Generate a Physical Graph Template that takes no time
                        to run
  --app=APP             Force an app to be used in the Physical Graph. 0=Don't
                        force, 1=SleepApp, 2=SleepAndCopy
  -N PARTITIONS, --partitions=PARTITIONS
                        Number of partitions to generate
  -i ISLANDS, --islands=ISLANDS
                        Number of islands to use during the partitioning
  -a ALGO, --algorithm=ALGO
                        algorithm used to do the partitioning
  -A ALGO_PARAMS, --algorithm-param=ALGO_PARAMS
                        Extra name=value parameters used by the algorithms
                        (algorithm-specific)
```

10.2.16 Command: dlg version

Help output:

```
Version: 1.0.0
Git version: Unknown
```

CHAPTER ELEVEN

API DOCUMENTATION

The following is an extract of the most important parts of the API of DALiuGE. For a complete reference please go to the source code.

11.1 dlg

Contents

11.1.1 dlg.ddap_protocol

`class dlg.ddap_protocol.`**`AppDROPStates`**

An enumeration of the different execution states an AppDROP can be found in. AppDROPs start in the NOT_RUN state, and move to the RUNNING state when they are started. Depending on the execution result they eventually move to the FINISHED or ERROR state.

`class dlg.ddap_protocol.`**`ChecksumTypes`**

An enumeration of different methods to calculate the checksum of a piece of data. DROPs (in certain conditions) calculate and keep the checksum of the data they represent, and therefore also know the method used to calculate it.

`class dlg.ddap_protocol.`**`DROPLinkType`**

An enumeration of the different relationships that can exist between DROPs.

Although not explicitly stated in this enumeration, each link type has a corresponding inverse. This way, if X is a consumer of Y, Y is an input of X. The full list is: * CONSUMER / INPUT * STREAMING_CONSUMER / STREAMING_INPUT * PRODUCER / OUTPUT * PARENT / CHILD

`class dlg.ddap_protocol.`**`DROPPhases`**

An enumeration of the different phases a DROP can be found in. Phases represent the persistence of the data associated to a DROP and the presence of replicas. Phases range from PLASMA (no replicas, volatile storage) to SOLID (fully backed up replica available).

`class dlg.ddap_protocol.`**`DROPRel`**(*lhs*, *rel*, *rhs*)

`property lhs`

Alias for field number 0

`property rel`

Alias for field number 1

`property rhs`

Alias for field number 2

`class dlg.ddap_protocol.`**`DROPStates`**

An enumeration of the different states a DROP can be found in. DROPs start in the INITIALIZED state, go optionally through WRITING and arrive to COMPLETED. Later, they transition through EXPIRED, eventually arriving to DELETED.

`class dlg.ddap_protocol.`**`ExecutionMode`**

Execution modes for a DROP. DROP means that a DROP will trigger its consumers automatically when it becomes COMPLETED. EXTERNAL means that a DROP will *not* trigger its consumers automatically, and instead this should be done by an external entity, probably by subscribing to changes on the DROP's status.

This value exists per DROP, and therefore we can achieve a mixed execution mode for the entire graph, where some DROPs trigger automatically their consumers, while others must be manually executed from the outside.

Note that if all DROPs in a graph have ExecutionMode == DROP it means that the graph effectively drives its own execution without external intervention.

11.1.2 dlg.drop

Module containing the core DROP classes.

`class dlg.drop.`**`AbstractDROP`**(*oid*, *uid*, ***kwargs*)

Base class for all DROP implementations.

A DROP is a representation of a piece of data. DROPs are created, written once, potentially read many times, and they finally potentially expire and get deleted. Subclasses implement different storage mechanisms to hold the data represented by the DROP.

If the data represented by this DROP is written *through* this object (i.e., calling the `write` method), this DROP will keep track of the data's size and checksum. If the data is written externally, the size and checksum can be fed into this object for future reference.

DROPs can have consumers attached to them. 'Normal' consumers will wait until the DROP they 'consume' (their 'input') moves to the COMPLETED state and then will consume it, most typically by opening it and reading its contents, but any other operation could also be performed. How the consumption is triggered depends on the producer's *executionMode* flag, which dictates whether it should trigger the consumption itself or if it should be manually triggered by an external entity. On the other hand, streaming consumers receive the data that is written into its input *as it gets written*. This mechanism is driven always by the DROP that acts as a streaming input. Apart from receiving the data as it gets written into the DROP, streaming consumers are also notified when the DROPs moves to the COMPLETED state, at which point no more data should be expected to arrive at the consumer side.

DROPs' data can be expired automatically by the system after the DROP has transitioned to the COMPLETED state if they are created by a DROP Manager. Expiration can either be triggered by an interval relative to the creation time of the DROP (via the *lifespan* keyword), or by specifying that the DROP should be expired after all its consumers have finished (via the *expireAfterUse* keyword). These two methods are mutually exclusive. If none is specified no expiration occurs.

`addConsumer`(*consumer*, *back=True*)

Adds a consumer to this DROP.

Consumers are normally (but not necessarily) AppDROPs that get notified when this DROP moves into the COMPLETED or ERROR states. This is done by firing an event of type *dropCompleted* to which the consumer subscribes to.

This is one of the key mechanisms by which the DROP graph is executed automatically. If AppDROP B consumes DROP A, then as soon as A transitions to COMPLETED B will be notified and will probably start its execution.

`addProducer`(*producer*, *back=True*)

Adds a producer to this DROP.

Producers are AppDROPs that write into this DROP; from the producers' point of view, this DROP is one of its many outputs.

When a producer has finished its execution, this DROP will be notified via the self.producerFinished() method.

`addStreamingConsumer`(*streamingConsumer*, *back=True*)

Adds a streaming consumer to this DROP.

Streaming consumers are AppDROPs that receive the data written into this DROP *as it gets written*, and therefore do not need to wait until this DROP has been moved to the COMPLETED state.

`autofill_environment_variables()`

Runs through all parameters here, fetching those which match the env-var syntax when discovered.

`cancel()`

Moves this drop to the CANCELLED state closing any writers we opened

`commit()`

Generates the MerkleRoot of this DROP Should only be called once this DROP is completed.

`completedrop()`

Builds final reproducibility data for this drop and fires a 'dropComplete' event. This should be called once a drop is finished in success or error :return:

`property consumers`

The list of 'normal' consumers held by this DROP.

See
self.addConsumer()

`dropReproComplete`(*uid*, *reprodata*)

Callback invoved when a DROP with UID *uid* has finishing processing its reproducibility information. Importantly, this is independent of that drop being completed.

property executionMode

The execution mode of this DROP. If *ExecutionMode.DROP* it means that this DROP will automatically trigger the execution of all its consumers. If *ExecutionMode.EXTERNAL* it means that this DROP will *not* trigger its consumers, and therefore an external entity will have to do it.

generate_merkle_data()

Provides a serialized summary of data as a list. Fields constitute a single entry in this list. Wraps several methods dependent on this DROPs reproducibility level Some of these are abstract. :return: A dictionary of elements constituting a summary of this drop

generate_recompute_data()

Provides a dictionary containing recompute data. At runtime, recomputing, like repeating and rerunning, by default, only shows success or failure. We anticipate that any further implemented behaviour be done at a lower class. :return: A dictionary containing runtime exclusive recompute values.

generate_repeat_data()

Provides a list of Repeat data. At runtime, repeating, like rerunning only requires execution success or failure. :return: A dictionary containing runtime exclusive repetition values.

generate_replicate_comp_data()

Provides a list of computational replication data. This is by definition a merging of both reproduction and recompute data :return: A dictionary containing runtime exclusive computational replication data.

generate_replicate_sci_data()

Provides a list of scientific replication data. This is by definition a merging of both reproduction and rerun data :return: A dictionary containing runtime exclusive scientific replication data.

generate_replicate_total_data()

Provides a list of total replication data. This is by definition a merging of reproduction and repetition data :return: A dictionary containing runtime exclusive total replication data.

generate_reproduce_data()

Provides a list of Reproducibility data (specifically). The default behaviour is to return nothing. Per-class behaviour is to be achieved by overriding this method. :return: A dictionary containing runtime exclusive reproducibility data.

generate_rerun_data()

Provides a serailized list of Rerun data. At runtime, Rerunning only requires execution success or failure. :return: A dictionary containing rerun values

get_consumers_nodes()

Gets the physical node address(s) of the consumer of this drop.

get_environment_variable(*key: str*)

Expects keys of the form $store_name.var_name $store_name.var_name.sub_var_name will query store_name for var_name.sub_var_name

get_environment_variables(*keys: list*)

Expects multiple instances of the single key form

handleEvent(*e*)

Handles the arrival of a new event. Events are delivered from those objects this DROP is subscribed to.

handleInterest(*drop*)

Main mechanism through which a DROP handles its interest in a second DROP it isn't directly related to.

A call to this method should be expected for each DROP this DROP is interested in. The default implementation does nothing, but implementations are free to perform any action, such as subscribing to events or storing information.

At this layer only the handling of such an interest exists. The expression of such interest, and the invocation of this method wherever necessary, is currently left as a responsibility of the entity creating the DROPs. In the case of a Session in a DROPManager for example this step would be performed using deployment-time information contained in the dropspec dictionaries held in the session.

`initialize`(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

`isCompleted`()

Checks whether this DROP is currently in the COMPLETED state or not

`property oid`

The DROP's Object ID (OID). OIDs are unique identifiers given to semantically different DROPs (and by consequence the data they represent). This means that different DROPs that point to the same data semantically speaking, either in the same or in a different storage, will share the same OID.

`property parent`

The DROP that acts as the parent of the current one. This parent/child relationship is created by ContainerDROPs, which are a specific kind of DROP.

`property persist`

Whether this DROP should be considered persisted after completion

`property phase`

This DROP's phase. The phase indicates the resilience of a DROP.

`producerFinished`(*uid*, *drop_state*)

Method called each time one of the producers of this DROP finishes its execution. Once all producers have finished this DROP moves to the COMPLETED state (or to ERROR if one of the producers is on the ERROR state).

This is one of the key mechanisms through which the execution of a DROP graph is accomplished. If AppDROP A produces DROP B, as soon as A finishes its execution B will be notified and will move itself to COMPLETED.

`property producers`

The list of producers that write to this DROP

See
self.addProducer()

`setCompleted`()

Moves this DROP to the COMPLETED state. This can be used when not all the expected data has arrived for a given DROP, but it should still be moved to COMPLETED, or when the expected amount of data held by a DROP is not known in advanced.

`setError`()

Moves this DROP to the ERROR state.

property size

The size of the data pointed by this DROP. Its value is automatically calculated if the data was actually written through this DROP (using the *self.write()* method directly or indirectly). In the case that the data has been externally written, the size can be set externally after the DROP has been moved to COMPLETED or beyond.

skip()

Moves this drop to the SKIPPED state closing any writers we opened

property status

The current status of this DROP.

property streamingConsumers

The list of 'streaming' consumers held by this DROP.

See
self.addStreamingConsumer()

property uid

The DROP's Unique ID (UID). Unlike the OID, the UID is globally different for all DROP instances, regardless of the data they point to.

class dlg.drop.**ListAsDict**(*my_set*)

A list that adds drop UIDs to a set as they get appended to the list

append(*drop*)

Append object to the end of the list.

11.1.3 dlg.droputils

Utility methods and classes to be used when interacting with DROPs

class dlg.droputils.**DROPFile**(*drop*)

A file-like object (currently only supporting the read() operation, more to be added in the future) that wraps the DROP given at construction time.

Depending on the underlying storage of the data the file-like object returned by this method will directly access the data pointed by the DROP if possible, or will access it through the DROP methods instead.

Objects of this class will automatically close themselves when no referenced anymore (i.e., when __del__ is called), but users should still try to invoke *close()* eagerly to free underlying resources.

Objects of this class can also be used in a *with* context.

class dlg.droputils.**DROPWaiterCtx**(*test*, *drops*, *timeout=1*, *expected_states=[]*)

Class used by unit tests to trigger the execution of a physical graph and wait until the given set of DROPs have reached its COMPLETED status.

It does so by appending an EvtConsumer consumer to each DROP before they are used in the execution, and finally checking that the events have been set. It should be used like this inside a test class:

```
# There is a physical graph that looks like: a -> b -> c
with DROPWaiterCtx(self, c):
    a.write('a')
    a.setCompleted()
```

class dlg.droputils.**EvtConsumer**(*evt*, *expected_states=[]*)

Small utility class that sets the internal flag of the given threading.Event object when consuming a DROP. Used throughout the tests as a barrier to wait until all DROPs of a given graph have executed.

dlg.droputils.**allDropContents**(*drop*, *bufsize=65536*) → bytes

Returns all the data contained in a given DROP

dlg.droputils.**breadFirstTraverse**(*toVisit*)

Breadth-first iterator for a DROP graph.

This iterator yields a tuple where the first item is the node being visited, and the second is a list of nodes that will be visited subsequently. Callers can alter this list in order to remove certain nodes from the graph traversal process.

This implementation is non-recursive.

dlg.droputils.**copyDropContents**(*source:* DataDROP, *target:* DataDROP, *bufsize: int = 65536*)

Manually copies data from one DROP into another, in bufsize steps

dlg.droputils.**depthFirstTraverse**(*node:* AbstractDROP, *visited=[]*)

Depth-first iterator for a DROP graph.

This iterator yields a tuple where the first item is the node being visited, and the second is a list of nodes that will be visited subsequently. Callers can alter this list in order to remove certain nodes from the graph traversal process.

This implementation is recursive.

dlg.droputils.**getDownstreamObjects**(*drop*)

Returns a list of all direct “downstream” DROPs for the given DROP. A DROP A is “downstream” with respect to DROP B if any of the following conditions are true: * A is an output of B (therefore B is an AppDROP) * A is a normal or streaming consumer of B (and A is therefore an AppDROP)

In practice if A is a downstream DROP of B means that it cannot advance to the COMPLETED state until B does so.

dlg.droputils.**getLeafNodes**(*drops*)

Returns a list of all the “leaf nodes” of the graph pointed by *drops*. *drops* is either a single DROP, or a list of DROPs.

dlg.droputils.**getUpstreamObjects**(*drop:* AbstractDROP)

Returns a list of all direct “upstream” DROPs for the given+ DROP. An DROP A is “upstream” with respect to DROP B if any of the following conditions are true:

- A is a producer of B (therefore A is an AppDROP)
- A is a normal or streaming input of B (and B is therefore an AppDROP)

In practice if A is an upstream DROP of B means that it must be moved to the COMPLETED state before B can do so.

dlg.droputils.**has_path**(*x*)

Returns *True* if *x* has a *path* attribute

dlg.droputils.**listify**(*o*)

If *o* is already a list return it as is; if *o* is a tuple returns a list containing the elements contained in the tuple; otherwise returns a list with *o* being its only element

dlg.droputils.**load_npy**(*drop:* DataDROP, *allow_pickle=False*) → <MagicMock id='140665414711696'>

Loads a numpy ndarray from a drop in npy format `dlg.droputils.load_pickle`(*drop: DataDROP*) → Any

Loads a pkl formatted data object stored in a DataDROP. Note: does not support streaming mode.

`dlg.droputils.replace_dataurl_placeholders`(*cmd*, *inputs*, *outputs*)

Replaces any placeholder found in `cmd` with the dataURL property of the respective input or output Drop from `inputs` or `outputs`. Placeholders have the different formats:

- `%iDataURLN`, with N starting from 0, indicates the path of the N-th element from the `inputs` argument; likewise for `%oDataURLN`.
- `%iDataURL[X]` indicates the path of the input with UID `X`; likewise for `%oDataURL[X]`.

`dlg.droputils.replace_path_placeholders`(*cmd*, *inputs*, *outputs*)

Replaces any placeholder found in `cmd` with the path of the respective input or output Drop from `inputs` or `outputs`. Placeholders have the different formats:

- `%iN`, with N starting from 0, indicates the path of the N-th element from the `inputs` argument; likewise for `%oN`.
- `%i[X]` indicates the path of the input with UID `X`; likewise for `%o[X]`.

`dlg.droputils.save_npy`(*drop: DataDROP*, *ndarray: <MagicMock id='140665415708880'>*, *allow_pickle=False*)

Saves a numpy ndarray to a drop in npy format

`dlg.droputils.save_pickle`(*drop: DataDROP*, *data: Any*)

Saves a python object in pkl format

11.1.4 dlg.event

`class dlg.event.Event`(*type: str*)

An event sent through the DALiuGE framework.

Events have at least a field describing the type of event they are (instead of having subclasses of the *Event* class), and therefore this class makes sure that at least that field exists. Any other piece of information can be attached to individual instances of this class, depending on the event type.

`class dlg.event.EventFirer`

An object that fires events.

Objects that have an interest on receiving events from this object subscribe to it via the *subscribe* method; likewise they can unsubscribe from it via the *unsubscribe* method. Events are handled to the listeners by calling their *handleEvent* method with the event as its sole argument.

Listeners can specify the type of event they listen to at subscription time, or can also prefer to receive all events fired by this object if they wish so.

`subscribe`(*listener: EventHandler*, *eventType: Optional[str] = None*)

Subscribes *listener* to events fired by this object. If *eventType* is not *None* then *listener* will only receive events of *eventType* that originate from this object, otherwise it will receive all events.

`unsubscribe`(*listener: EventHandler*, *eventType: Optional[str] = None*)

Unsubscribes *listener* from events fired by this object.

`class dlg.event.EventHandler`

11.1.5 dlg.graph_loader

Module containing functions to load a fully-functional DROP graph from its full JSON representation.

`dlg.graph_loader.`**`addLink`**`(`*linkType*, *lhDropSpec*, *rhOID*, *force=False*`)`

Adds a link from *lhDropSpec* to point to *rhOID*. The link type (e.g., a consumer) is signaled by *linkType*.

`dlg.graph_loader.`**`loadDropSpecs`**`(`*dropSpecList*`)`

Loads the DROP definitions from *dropSpectList*, checks that the DROPs are correctly specified, and return a dictionary containing all DROP specifications (i.e., a dictionary of dictionaries) keyed on the OID of each DROP. Unlike *readObjectGraph* and *readObjectGraphS*, this method doesn't actually create the DROPs themselves.

Slices off graph-wise reproducibility data for later use

11.1.6 dlg.rpc

RPC support for DALiuGE

This module contains all client and server RPC classes for all the different technologies we support.

`class` `dlg.rpc.`**`DropProxy`**`(`*rpc_client*, *hostname*, *port*, *sessionId*, *uid*`)`

A proxy to a remote drop.

It forwards attribute requests and procedure calls through the given RPC client.

`dlg.rpc.`**`RPCClient`**

alias of *`ZeroRPCClient`*

`class` `dlg.rpc.`**`RPCClientBase`**

Base class for all RPC clients

`class` `dlg.rpc.`**`RPCObject`**

Base class for all RCP clients and server

`dlg.rpc.`**`RPCServer`**

alias of *`ZeroRPCServer`*

`class` `dlg.rpc.`**`RPCServerBase`**`(`*host*, *port*`)`

Base class for all RPC server

`class` `dlg.rpc.`**`ZeroRPCClient`**`(`**args*, ***kwargs*`)`

ZeroRPC client support

`class` **`request`**`(`*method*, *args*, *queue*`)`

`property` **`args`**

Alias for field number 1

`property` **`method`**

Alias for field number 0

`property` **`queue`**

Alias for field number 2

`class` **`response`**`(`*value*, *is_exception*`)`

`property` **`is_exception`**

Alias for field number 1

property value

Alias for field number 0

class `dlg.rpc.`**`ZeroRPCServer`**(*host*, *port*)

ZeroRPC server support

11.1.7 dlg.runtime.delayed

`dlg.runtime.`**`delayed`**(*x*, **args*, ***kwargs*)

Like dask.delayed, but quietly swallowing anything other than *nout*

11.1.8 dlg.utils

Module containing miscellaneous utility classes and functions.

class `dlg.utils.`**`ExistingProcess`**(*pid*)

A Popen-like class around an existing process

`kill`()

Send a KILL signal

`poll`()

Returns an exit status if the process finished, None if it exists

`terminate`()

Send a TERM signal

`wait`()

Wait until the process finishes

class `dlg.utils.`**`ZlibUncompressedStream`**(*content*)

A class that reads gzip-compressed content and returns uncompressed content each time its read() method is called.

`dlg.utils.`**`browse_service`**(*zc*, *service_type_name*, *protocol*, *callback*)

ZeroConf: Browse for services based on service type and protocol

callback signature: callback(zeroconf, service_type, name, state_change)

zeroconf: ZeroConf object service_type: zeroconf service name: service name state_change: ServiceStateChange type (Added, Removed)

Returns ZeroConf object

`dlg.utils.`**`createDirIfMissing`**(*path*)

Creates the given directory if it doesn't exist

`dlg.utils.`**`deregister_service`**(*zc*, *info*)

ZeroConf: Deregister service

`dlg.utils.`**`escapeQuotes`**(*s*, *singleQuotes=True*, *doubleQuotes=True*)

Escapes single and double quotes in a string. Useful to include commands in a shell invocation or similar.

`dlg.utils.`**`fname_to_pipname`**(*fname*)

Converts a graph filename (extension .json or .graph) to its "pipeline" name (the basename without the extension).

dlg.utils.**getDlgDir**()

Returns the root of the directory structure used by the DALiuGE framework at runtime.

dlg.utils.**getDlgLogsDir**()

Returns the location of the directory used by the DALiuGE framework to store its logs. If *createIfMissing* is True, the directory will be created if it currently doesn't exist

dlg.utils.**getDlgPath**()

Returns the location of the directory used by the DALiuGE framework to look for additional code. If *createIfMissing* is True, the directory will be created if it currently doesn't exist

dlg.utils.**getDlgPidDir**()

Returns the location of the directory used by the DALiuGE framework to store its PIDs. If *createIfMissing* is True, the directory will be created if it currently doesn't exist

dlg.utils.**getDlgVariable**(*key: str*)

Queries environment for variables assumed to start with '*DLG*'. Special case for DLG_ROOT, since this is easily identifiable.

dlg.utils.**getDlgWorkDir**()

Returns the location of the directory used by the DALiuGE framework to store results. If *createIfMissing* is True, the directory will be created if it currently doesn't exist

dlg.utils.**get_all_ipv4_addresses**()

Get a list of all IPv4 interfaces found in this computer

dlg.utils.**get_local_ip_addr**()

Enumerate all interfaces and return bound IP addresses (exclude localhost)

dlg.utils.**get_symbol**(*name*)

Gets the global symbol `name`, which is an "absolute path" to a python name in the form of `pkg.subpkg.subpkg.module.name`

dlg.utils.**isabs**(*path*)

Like os.path.isabs, but handles None

dlg.utils.**object_tracking**(*name*)

Returns a decorator that helps classes track which object is currently under execution. This is done via a thread local object, which can be accessed via the 'tlocal' attribute of the returned decorator.

dlg.utils.**prepare_sql**(*sql*, *paramstyle*, *data=()*) → Tuple[str, dict]

Prepares the given SQL statement for proper execution depending on the parameter style supported by the database driver. For this the SQL statement must be written using the "{X}" or "{}" placeholders in place for each, parameter which is a style-agnostic parameter notation.

This method returns a tuple containing the prepared SQL statement and the values to be bound into the query as required by the driver.

dlg.utils.**register_service**(*zc*, *service_type_name*, *service_name*, *ipaddr*, *port*, *protocol='tcp'*)

ZeroConf: Register service type, protocol, ipaddr and port

Returns ZeroConf object and ServiceInfo object

dlg.utils.**timed_import**(*module_name*)

Imports *module_name* and log how long it took to import it `dlg.utils.to_externally_contactable_host(host, prefer_local=False)`

Turns *host*, which is an address used to bind a local service, into a host that can be used to externally contact that service.

This should be used when there is no other way to find out how a client to that service is going to connect to it.

`dlg.utils.zmq_safe(host_or_addr)`

Converts *host_or_addr* to a format that is safe for ZMQ to use

11.2 dlg.manager

This package contains all python modules implementing the DROP Manager concepts, including their external interface, a web UI and a client

Contents

11.2.1 dlg.manager.session

Module containing the logic of a session – a given graph execution

11.2.2 dlg.manager.drop_manager

Module containing the base interface for all DROP managers.

`class dlg.manager.drop_manager.DROPManager`

Base class for all DROPManagers.

A DROPManager, as the name states, manages the creation and execution of DROPs. In order to support parallel DROP graphs execution, a DROPManager separates them into "sessions".

Sessions follow a simple lifecycle:

- They are created in the PRISTINE status
- One or more graph specifications are appended to them, which can also be linked together, building up the final graph specification. While building the graph the session is in the BUILDING status.
- Once all graph specifications have been appended and linked together, the graph is deployed, meaning that the DROPs are effectively created. During this process the session transitions between the DEPLOYING and RUNNING states.

- One all DROPs contained in a session have transitioned to COMPLETED (or ERROR, if there has been an error during the execution) the session moves to FINISHED.

Graph specifications are currently accepted in the form of a list of dictionaries, where each dictionary is a DROP specification. A DROP specification in turn consists on key/value pairs in the dictionary which state the type of DROP, some key parameters, and instance-specific parameters as well used to create the DROP.

abstract addGraphSpec(*sessionId*, *graphSpec*)

Adds a graph specification *graphSpec* (i.e., a description of the DROPs that should be created) to the current graph specification held by session *sessionId*.

abstract cancelSession(*sessionId*)

Cancels the session *sessionId*

abstract createSession(*sessionId*)

Creates a session on this DROPManager with id *sessionId*. A session represents an isolated DROP graph execution.

abstract deploySession(*sessionId*, *completedDrops=[]*)

Deploys the graph specification held by session *sessionId*, effectively creating all DROPs, linking them together, and moving those whose UID is in *completedDrops* to the COMPLETED state.

abstract destroySession(*sessionId*)

Destroys the session *sessionId*

abstract getGraph(*sessionId*)

Returns a specification of the graph currently held by session *sessionId*.

abstract getGraphReproData(*sessionId*)

Returns the graph-wide reproducibility data for session *sessionId*

abstract getGraphSize(*sessionId*)

Returns the number of drops contained in the physical graph attached to `sessionId`.

abstract getGraphStatus(*sessionId*)

Returns the status of the graph being executed in session *sessionId*.

abstract getSessionIds()

Returns the IDs of the sessions currently held by this DROPManager.

abstract getSessionReproStatus(*sessionId*)

Returns the reproducibility status of the session *sessionId*. Not guaranteed to be identical to the usual SessionStatus.

abstract getSessionStatus(*sessionId*)

Returns the status of the session *sessionId*.

11.2.3 dlg.manager.node_manager

Module containing the NodeManager, which directly manages DROP instances, and thus represents the bottom of the DROP management hierarchy.

class dlg.manager.node_manager.**ErrorStatusListener**(*session*, *error_listener*)

An event listener that passes down the erroneous drop to an error handler

dlg.manager.node_manager.**EventMixIn**

alias of *ZMQPubSubMixIn*

class dlg.manager.node_manager.**NodeManager**(*host=None*, *rpc_port=6666*, *events_port=5555*, **args*, ***kwargs*)

class dlg.manager.node_manager.**NodeManagerBase**(*dlm_check_period=0*, *dlm_cleanup_period=0*, *dlm_enable_replication=False*, *dlgPath=None*, *error_listener=None*, *event_listeners=[]*, *max_threads=0*, *logdir='/home/docs/dlg/logs'*)

Base class for a DROPManager that creates and holds references to DROPs.

A NodeManagerBase is the ultimate responsible of handling DROPs. It does so not directly, but via Sessions, which represent and encapsulate separate, independent DROP graph executions. All DROPs created by the different Sessions are also given to a common DataLifecycleManager, which takes care of expiring them when needed and replicating them.

Since a NodeManagerBase can handle more than one session, in principle only one NodeManagerBase is needed for each computing node, thus its name.

addGraphSpec(*sessionId*, *graphSpec*)

Adds a graph specification *graphSpec* (i.e., a description of the DROPs that should be created) to the current graph specification held by session *sessionId*.

cancelSession(*sessionId*)

Cancels the session *sessionId*

createSession(*sessionId*)

Creates a session on this DROPManager with id *sessionId*. A session represents an isolated DROP graph execution.

deliver_event(*evt*)

Method called by subclasses when a new event has arrived through the subscription mechanism.

deploySession(*sessionId*, *completedDrops=[]*)

Deploys the graph specification held by session *sessionId*, effectively creating all DROPs, linking them together, and moving those whose UID is in *completedDrops* to the COMPLETED state.

destroySession(*sessionId*)

Destroys the session *sessionId*

getGraph(*sessionId*)

Returns a specification of the graph currently held by session *sessionId*.

getGraphReproData(*sessionId*)

Returns the graph-wide reproducibility data for session *sessionId*

getGraphSize(*sessionId*)

Returns the number of drops contained in the physical graph attached to `sessionId`.

getGraphStatus(*sessionId*)

Returns the status of the graph being executed in session *sessionId*.

getSessionIds()

Returns the IDs of the sessions currently held by this DROPManager.

getSessionReproStatus(*sessionId*)

Returns the reproducibility status of the session *sessionId*. Not guaranteed to be identical to the usual SessionStatus.

getSessionStatus(*sessionId*)

Returns the status of the session *sessionId*.

class dlg.manager.node_manager.**RpcMixIn**(**args*, ***kwargs*)

class dlg.manager.node_manager.**ZMQPubSubMixIn**(*host*, *events_port*)

ZeroMQ-based event publisher and subscriber.

Event publishing and event reception are done in their own separate threads, where the externally-facing ZeroMQ sockets are created and used.

Events to be published are fed into the publishing thread via a safe-thread Queue object (self._events_out), enabling any local thread to publish events without having to worry about ZeroMQ thread-safeness.

The event reception thread not only *receives* events, but also updates the subscription socket to connect to new peers. These updates are fed via a Queue object (self._subscriptions), enabling any local thread to indicate a new peer to subscribe to in a thread-safe manner.

Note that we investigated not using Queue objects to communicate between threads, and use inproc:// ZeroMQ sockets instead. This works, but at a cost: all threads putting values into these sockets would need to check, each time they use a socket in any manner, if the Context object is still valid and hasn't been closed (or alternatively if self._pubsub_running is still True). Our experience with this alternative was not satisfactory, and therefore we went for a Queue-based thread communication model, making the handling of ZeroMQ resources simpler.

class **subscription**(*endpoint*, *finished_evt*)

property **endpoint**

Alias for field number 0

property **finished_evt**

Alias for field number 1

11.2.4 dlg.manager.composite_manager

class dlg.manager.composite_manager.**CompositeManager**(*dmPort*, *partitionAttr*, *subDmId*, *dmHosts=[]*, *pkeyPath=None*, *dmCheckTimeout=10*)

A DROPManager that in turn manages DROPManagers (sigh…).

DROP Managers form a hierarchy where those at the bottom actually hold DROPs while those in the levels above rely commands and aggregate results, making the system more manageable and scalable. The CompositeManager class implements the upper part of this hierarchy in a generic way by holding references to a number of sub-DROPManagers and communicating with them to complete each operation. The only assumption about sub-DROPManagers is that they obey the DROPManager interface, and therefore this CompositeManager class allows for multiple levels of hierarchy seamlessly.

Having different levels of Data Management hierarchy implies that the physical graph that is fed into the hierarchy needs to be partitioned at each level (except at the bottom of the hierarchy) in order to place each DROP in its correct place. The attribute used by a particular CompositeManager to partition the graph (from its graphSpec) is given at construction time.

addGraphSpec(*sessionId*, *graphSpec*)

Adds a graph specification *graphSpec* (i.e., a description of the DROPs that should be created) to the current graph specification held by session *sessionId*.

cancelSession(*sessionId*)

Cancels a session in all underlying DMs.

createSession(*sessionId*)

Creates a session in all underlying DMs.

deploySession(*sessionId*, *completedDrops=[]*)

Deploys the graph specification held by session *sessionId*, effectively creating all DROPs, linking them together, and moving those whose UID is in *completedDrops* to the COMPLETED state.

destroySession(*sessionId*)

Destroy a session in all underlying DMs.

getGraph(*sessionId*)

Returns a specification of the graph currently held by session *sessionId*.

getGraphSize(*sessionId*)

Returns the number of drops contained in the physical graph attached to `sessionId`.

getGraphStatus(*sessionId*)

Returns the status of the graph being executed in session *sessionId*.

getSessionIds()

Returns the IDs of the sessions currently held by this DROPManager.

getSessionStatus(*sessionId*)

Returns the status of the session *sessionId*.

replicate(*sessionId*, *f*, *action*, *collect=None*, *iterable=None*, *port=None*)

Replicates the given function call on each of the underlying drop managers

class dlg.manager.composite_manager.**DataIslandManager**(*dmHosts=[]*, *pkeyPath=None*, *dmCheckTimeout=10*)

The DataIslandManager, which manages a number of NodeManagers.

class dlg.manager.composite_manager.**MasterManager**(*dmHosts=[]*, *pkeyPath=None*, *dmCheckTimeout=10*)

The MasterManager, which manages a number of DataIslandManagers.

dlg.manager.composite_manager.**sanitize_link**(*link*)

Links can now be dictionaries, but we only need the key.

11.2.5 dlg.manager.rest

Module containing the REST layer that exposes the methods of the different Data Managers (DROPManager and DataIslandManager) to the outside world.

class dlg.manager.rest.**CompositeManagerRestServer**(*dm*, *maxreqsize=10*)

A REST server for DataIslandManagers. It includes mappings for DIM-specific methods.

initializeSpecifics(*app*)

Methods through which subclasses can initialize other mappings on top of the default ones and perform other DataManager-specific actions. The default implementation does nothing.

class dlg.manager.rest.**ManagerRestServer**(*dm*, *maxreqsize=10*)

An object that wraps a DataManager and exposes its methods via a REST interface. The server is started via the *start* method in a separate thread and runs until the process is shut down.

This REST server currently also serves HTML pages in some of its methods (i.e. those not under /api).

initializeSpecifics(*app*)

Methods through which subclasses can initialize other mappings on top of the default ones and perform other DataManager-specific actions. The default implementation does nothing.

class dlg.manager.rest.**MasterManagerRestServer**(*dm*, *maxreqsize=10*)

initializeSpecifics(*app*)

Methods through which subclasses can initialize other mappings on top of the default ones and perform other DataManager-specific actions. The default implementation does nothing.

class dlg.manager.rest.**NMRestServer**(*dm*, *maxreqsize=10*)

A REST server for NodeManagers. It includes mappings for NM-specific methods and the mapping for the main visualization HTML pages.

initializeSpecifics(*app*)

Methods through which subclasses can initialize other mappings on top of the default ones and perform other DataManager-specific actions. The default implementation does nothing.

11.2.6 dlg.manager.client

Backwards compatibility for client

11.3 dlg.apps

Contents

11.3.1 dlg.apps.app_base

`class dlg.apps.app_base.AppDROP`(*oid*, *uid*, ***kwargs*)

An AppDROP is a DROP representing an application that reads data from one or more DataDROPs (its inputs), and writes data onto one or more DataDROPs (its outputs).

AppDROPs accept two different kind of inputs: "normal" and "streaming" inputs. Normal inputs are DataDROPs that must be on the COMPLETED state (and therefore their data must be fully written) before this application is run, while streaming inputs are DataDROPs that feed chunks of data into this application as the data gets written into them.

This class contains two methods that need to be overwritten by subclasses: *dropCompleted*, invoked when input DataDROPs move to COMPLETED, and *dataWritten*, invoked with the data coming from streaming inputs.

How and when applications are executed is completely up to the app component developer, and is not enforced by this base class. Some applications might need to be run at *initialize* time, while other might start during the first invocation of *dataWritten*. A common scenario anyway is to start an application only after all its inputs have moved to COMPLETED (implying that none of them is an streaming input); for these cases see the *BarrierAppDROP*.

`cancel`()

Moves this application drop to its CANCELLED state

`dataWritten`(*uid*, *data*)

Callback invoked when *data* has been written into the DROP with UID *uid* (which is one of the streaming inputs of this AppDROP). By default no action is performed

`dropCompleted`(*uid*, *drop_state*)

Callback invoked when the DROP with UID *uid* (which is either a normal or a streaming input of this AppDROP) has moved to the COMPLETED or ERROR state. By default no action is performed.

`property execStatus`

The execution status of this AppDROP

`handleEvent`(*e*)

Handles the arrival of a new event. Events are delivered from those objects this DROP is subscribed to.

`initialize`(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

`property inputs: List[DataDROP]`

The list of inputs set into this AppDROP

`property outputs: List[DataDROP]`

The list of outputs set into this AppDROP

`skip`()

Moves this application drop to its SKIPPED state

`property streamingInputs: List[DataDROP]`

The list of streaming inputs set into this AppDROP

class `dlg.apps.app_base.`**`BarrierAppDROP`**(*oid*, *uid*, ***kwargs*)

A BarrierAppDROP is an InputFireAppDROP that waits for all its inputs to complete, effectively blocking the flow of the graph execution.

`initialize`(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

class `dlg.apps.app_base.`**`InputFiredAppDROP`**(*oid*, *uid*, ***kwargs*)

An InputFiredAppDROP accepts no streaming inputs and waits until a given amount of inputs (called *effective inputs*) have moved to COMPLETED to execute its 'run' method, which must be overwritten by subclasses. This way, this application allows to continue the execution of the graph given a minimum amount of inputs being ready. The transitions of subsequent inputs to the COMPLETED state have no effect.

Normally only one call to the *run* method will happen per application. However users can override this by specifying a different number of tries before finally giving up.

The amount of effective inputs must be less or equal to the amount of inputs added to this application once the graph is being executed. The special value of -1 means that all inputs are considered as effective, in which case this class acts as a BarrierAppDROP, effectively blocking until all its inputs have moved to the COMPLETED, SKIPPED or ERROR state. Setting this value to anything other than -1 or the number of inputs, results in late arriving inputs to be ignored, even if they would successfully finish. This requires careful implementation of the upstream and downstream apps to deal with this situation. It is only really useful to control a combination of maximum allowed execution time and acceptable number of completed inputs.

An input error threshold controls the behavior of the application given an error in one or more of its inputs (i.e., a DROP moving to the ERROR state). The threshold is a value within 0 and 100 that indicates the tolerance to erroneous effective inputs, and after which the application will not be run but moved to the ERROR state itself instead.

`dropCompleted`(*uid*, *drop_state*)

Callback invoked when the DROP with UID *uid* (which is either a normal or a streaming input of this AppDROP) has moved to the COMPLETED or ERROR state. By default no action is performed.

`execute`(*_send_notifications=True*)

Manually trigger the execution of this application.

This method is normally invoked internally when the application detects all its inputs are COMPLETED.

`exists`()

Returns *True* if the data represented by this DROP exists indeed in the underlying storage mechanism

`initialize`(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

`run`()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

11.3.2 dlg.apps.bash_shell_app

Module containing bash-related AppDrops

The module contains four classes that offer running bash commands in different execution modes; that is, in fully batch mode, or with its input and/or output as a stream of data to the previous/next application.

`class dlg.apps.bash_shell_app.`**`BashShellApp`**`(`*oid*, *uid*, ***kwargs*`)`

An app that runs a bash command in batch mode; that is, it waits until all its inputs are COMPLETED. It also *doesn't* output a stream of data; see StreamingOutputBashApp for those cases.

`run``()`

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

`class dlg.apps.bash_shell_app.`**`BashShellBase`**

Common class for BashShell apps. It simply requires a command to be specified.

`class dlg.apps.bash_shell_app.`**`StreamingInputBashApp`**`(`*oid*, *uid*, ***kwargs*`)`

An app that runs a bash command that consumes data from stdin.

The streaming of data that appears on stdin takes place outside the framework; what is streamed through the framework is the information needed to establish the streaming channel. This information is also used to kick this application off.

`class dlg.apps.bash_shell_app.`**`StreamingInputBashAppBase`**`(`*oid*, *uid*, ***kwargs*`)`

Base class for bash command applications that consume a stream of incoming data.

`dataWritten``(`*uid*, *data*`)`

Callback invoked when *data* has been written into the DROP with UID *uid* (which is one of the streaming inputs of this AppDROP). By default no action is performed

`dropCompleted``(`*uid*, *drop_state*`)`

Callback invoked when the DROP with UID *uid* (which is either a normal or a streaming input of this AppDROP) has moved to the COMPLETED or ERROR state. By default no action is performed.

`initialize``(`***kwargs*`)`

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

`class dlg.apps.bash_shell_app.`**`StreamingInputOutputBashApp`**`(`*oid*, *uid*, ***kwargs*`)`

Like StreamingInputBashApp, but its stdout is also a stream of data that is fed into the next application.

`class dlg.apps.bash_shell_app.`**`StreamingOutputBashApp`**`(`*oid*, *uid*, ***kwargs*`)`

Like BashShellApp, but its stdout is a stream of data that is fed into the next application.

`run``()`

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

`dlg.apps.bash_shell_app.`**`prepare_input_channel`**`(`*data*`)`

Prepares an input channel that will serve as the stdin of a bash command. Depending on the contents of `data` the channel will be a named pipe or a socket.

dlg.apps.bash_shell_app.**prepare_output_channel**(*this_node*, *out_drop*)

Prepares an output channel that will serve as the stdout of a bash command. Depending on the values of `this_node` and `out_drop` the channel will be a named pipe or a socket.

11.3.3 dlg.apps.branch

class dlg.apps.branch.**BranchAppDrop**(*oid*, *uid*, ***kwargs*)

A special kind of application with exactly two outputs. After normal execution, the application decides whether a certain condition is met. If the condition is met, the first output is considered as COMPLETED, while the other is moved to SKIPPED state, and vice-versa.

execute(*_send_notifications=True*)

Manually trigger the execution of this application.

This method is normally invoked internally when the application detects all its inputs are COMPLETED.

11.3.4 dlg.apps.constructs

class dlg.apps.constructs.**CommentDrop**(*oid*, *uid*, ***kwargs*)

This only exists to make sure we have a comment in the template palette

class dlg.apps.constructs.**DescriptionDrop**(*oid*, *uid*, ***kwargs*)

This only exists to make sure we have a description in the template palette

class dlg.apps.constructs.**ExclusiveForceDrop**(*oid*, *uid*, ***kwargs*)

This only exists to make sure we have an exclusive force node in the template palette

class dlg.apps.constructs.**GatherDrop**(*oid*, *uid*, ***kwargs*)

This only exists to make sure we have a GroupBy in the template palette

class dlg.apps.constructs.**GroupByDrop**(*oid*, *uid*, ***kwargs*)

This only exists to make sure we have a GroupBy in the template palette

class dlg.apps.constructs.**LoopDrop**(*oid*, *uid*, ***kwargs*)

This only exists to make sure we have a loop in the template palette

class dlg.apps.constructs.**MKNDrop**(*oid*, *uid*, ***kwargs*)

This only exists to make sure we have a MKN in the template palette

class dlg.apps.constructs.**ScatterDrop**(*oid*, *uid*, ***kwargs*)

This only exists to make sure we have a Scatter in the template palette

class dlg.apps.constructs.**SubGraphDrop**(*oid*, *uid*, ***kwargs*)

This only exists to make sure we have a SubGraph in the template palette

11.3.5 dlg.apps.dynlib

class dlg.apps.dynlib.**CDlgApp**

class dlg.apps.dynlib.**CDlgInput**

class dlg.apps.dynlib.**CDlgOutput**

class dlg.apps.dynlib.**CDlgStreamingInput**

class dlg.apps.dynlib.**DynlibApp**(*oid*, *uid*, ***kwargs*)

Loads a dynamic library into the current process and runs it

generate_recompute_data()

Provides a dictionary containing recompute data. At runtime, recomputing, like repeating and rerunning, by default, only shows success or failure. We anticipate that any further implemented behaviour be done at a lower class. :return: A dictionary containing runtime exclusive recompute values.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.dynlib.**DynlibProcApp**(*oid*, *uid*, ***kwargs*)

Loads a dynamic library in a different process and runs it

cancel()

Moves this application drop to its CANCELLED state

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.dynlib.**DynlibStreamApp**(*oid*, *uid*, ***kwargs*)

dataWritten(*uid*, *data*)

Callback invoked when *data* has been written into the DROP with UID *uid* (which is one of the streaming inputs of this AppDROP). By default no action is performed

dropCompleted(*uid*, *drop_state*)

Callback invoked when the DROP with UID *uid* (which is either a normal or a streaming input of this AppDROP) has moved to the COMPLETED or ERROR state. By default no action is performed.

generate_recompute_data()

Provides a dictionary containing recompute data. At runtime, recomputing, like repeating and rerunning, by default, only shows success or failure. We anticipate that any further implemented behaviour be done at a lower class. :return: A dictionary containing runtime exclusive recompute values.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

exception dlg.apps.dynlib.**FinishSubprocess**

exception dlg.apps.dynlib.**InvalidLibrary**

dlg.apps.dynlib.**get_from_subprocess**(*proc*, *q*)

Gets elements from the queue, checking that the process is still alive

dlg.apps.dynlib.**load_and_init**(*libname*, *oid*, *uid*, *params*)

Loads and initializes *libname* with the given parameters, prepares the corresponding C application structure, and returns both objects

dlg.apps.dynlib.**prepare_c_inputs**(*c_app*, *inputs*)

Converts all inputs to its C equivalents and sets them into *c_app*

dlg.apps.dynlib.**prepare_c_outputs**(*c_app*, *outputs*)

Converts all outputs to its C equivalents and sets them into *c_app*

dlg.apps.dynlib.**prepare_c_ranks**(*c_app*, *ranks*)

Convert the ranks list into its C equivalent and sets them to *c_app*

dlg.apps.dynlib.**run**(*lib*, *c_app*, *input_closers*)

Invokes the *run* method on *lib* with the given *c_app*. After completion, all opened file descriptors are closed.

11.3.6 dlg.apps.mpi

Module containing MPI application wrapping support

class dlg.apps.mpi.**MPIApp**(*oid*, *uid*, ***kwargs*)

An application drop representing an MPI job.

This application needs to be launched from within an MPI environment, and therefore the hosting NM must be part of an MPI communicator. This application uses MPI_Comm_Spawn to fire up the requested MPI application, which must *not* be aware of it having a parent. This drop will gather the individual exit codes from the launched applications and transition to ERROR if any of them did not exit cleanly, or to FINISHED if all of them finished successfully.

generate_recompute_data()

Provides a dictionary containing recompute data. At runtime, recomputing, like repeating and rerunning, by default, only shows success or failure. We anticipate that any further implemented behaviour be done at a lower class. :return: A dictionary containing runtime exclusive recompute values.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

11.3.7 dlg.apps.plasmaflight

class dlg.apps.plasmaflight.**PlasmaFlightClient**(*socket: str*, *scheme: str = 'grpc+tcp'*, *connection_args: Optional[dict] = None*)

Client for accessing plasma-backed arrow flight data server.

create(*object_id: <MagicMock name='mock.ObjectID' id='140665427112400'>*, *size: int*) → <MagicMock name='mock.PlasmaBuffer' id='140665427053456'>

Creates an empty plasma buffer

exists(*object_id: <MagicMock name='mock.ObjectID' id='140665427112400'>*, *owner: ~typing.Optional[str] = None*) → bool

Returns true if the remote plasmaflight server contains the plasma object.

get_buffer(*object_id: <MagicMock name='mock.ObjectID' id='140665427112400'>*, *owner: ~typing.Optional[str] = None*) → memoryview

Gets the plasma object from the local store if it's available, otherwise queries the plasmaflight owner for the object.

get_flight(*object_id: <MagicMock name='mock.ObjectID' id='140665427112400'>*, *location: str*) → <MagicMock name='mock.FlightStreamReader' id='140665427070672'>

Retreives an flight object stream

list_flights(*location: str*)

Retrieves a list of flights

put_raw_buffer(*data: memoryview*, *object_id: <MagicMock name='mock.ObjectID' id='140665427112400'>*)

Puts

seal(*object_id: <MagicMock name='mock.ObjectID' id='140665427112400'>*)

Seals the plasma buffer marking it as readonly

11.3.8 dlg.apps.pyfunc

Module implementing the PyFuncApp class

`class dlg.apps.pyfunc.DropParser(`*value*`)`

An enumeration.

`class dlg.apps.pyfunc.PyFuncApp(`*oid*, *uid*, ***kwargs*`)`

An application that wraps a simple python function.

The inputs of the application are treated as the arguments of the function. Conversely, the output of the function is treated as the output of the application. If the application has more than one output, the result of calling the function is treated as an iterable, with each individual object being written to its corresponding output.

Users indicate the function to be wrapped via the `func_name` parameter. In this case func_name needs to specify a funtion in the standard form

`module.function`

and the module needs to be accessible on the PYTHONPATH of the DALiuGE engine. Note that the engine is expanding the standard PYTHONPATH with DLG_ROOT/code. That directory is always available, even if the engine is running in a docker container.

Otherwise, users can also *send* over the python code using the `func_code` parameter. The code needs to be base64-encoded and produced with the marshal module of the same Python version used to run DALiuGE.

The positional onlyargs will be used in order of appearance.

`generate_recompute_data()`

Provides a dictionary containing recompute data. At runtime, recomputing, like repeating and rerunning, by default, only shows success or failure. We anticipate that any further implemented behaviour be done at a lower class. :return: A dictionary containing runtime exclusive recompute values.

`initialize(`***kwargs***`)`**

The initialization of a function component is mainly dealing with mapping inputs and provided applicationArgs to the function arguments. All of this should be driven by matching names.

`initialize_with_func_code()`

This function takes over if code is passed in through an argument.

`run()`

Function positional and keyword argument treatment:

Function arguments can be provided in four different ways: 1) Through an input port 2) By specifying ApplicationArgs (one for each argument) 3) Through defaults at the time of function definition

The priority follows the list above with input ports overruling the others. Function arguments in Python can be passed as positional, kw-value, positional only, kw-value only, and catch-all args and kwargs, which don't provide any hint about the names of accepted parameters. All of them are now supported. If positional arguments or kw-value arguments are provided by the user, but are not explicitely defined in the function signiture AND args and/or kwargs are allowed then these arguments are passed to the function. For args this is somewhat risky, since the order is relevant and in this code derived from the order defined in the graph (same order as defined in the component description).

Input ports will NOT be used by order (anymore), but by the name of the port. Since each input port requires an associated data drop, this provides a unique mapping. This also allows to pass values to any function argument through a port.

Function argument values as well as the function code can be provided in serialised (pickle) form by setting the 'pickle' flag. Note that this flag is valid for all arguments and the code (if specified) in a global way.

11.3.9 dlg.apps.dockerapp

Module containing docker-related applications and functions

`class dlg.apps.dockerapp.ContainerIpWaiter(drop)`

A class that remembers the target DROP's uid and containerIp properties when its internal event has been set, and returns them when waitForIp is called, which previously waits for the event to be set.

`class dlg.apps.dockerapp.DockerApp(oid, uid, **kwargs)`

A BarrierAppDROP that represents a process running in a container hosted by a local docker daemon. Depending on the host system, the docker daemon might be automatically activated when a client tries to connect to it via its unix socket (like with systemd) or it needs to be brought up prior to any client operation (upstart). In any case, if the daemon is not present, this class will raise exceptions whenever it tries to connect to the server to perform some operation.

Docker containers are built from docker images, which are pulled to the host where the docker daemon runs either explicitly (via *docker pull*) or less visibly (e.g., when running *docker run* using an image that has not been fetched yet). This DockerApp application will explicitly pull the image at *initialize* time, meaning that the docker images will become available at the time the physical graph (which this application is part of) is deployed. Docker containers also need a command to be run in them, which should be an available program inside the image. Optionally, users can provide a working directory (in the container) under which the command will run via the *workingDir* parameter.

Input and output

The inputs and outputs used by the dockerized application are made available by mapping host directories and files as "data volumes". Inputs are bound using their full path, but outputs are bound only up to their dirnames, because otherwise they would be created at container creation time by Docker. For example, the output /a/b/c will produce a binding to /dlg/a/b inside the docker container, where c will have to be written by the process running in the container.

Since the command to be run in the container receives most probably as arguments the paths of its inputs and outputs, and since these might not be known precisely until runtime, users should use placeholders for them in the command-line specification. Placeholders for input locations take the form of "%iX", where X starts from 0 and refers to the X-th filesystem-related input. Likewise, output locations are specified as "%oX". Alternatively, inputs and outputs can be referred to by their UIDs, in which case the placeholders will look like "%i[X]" and "%o[X]" respectively, where X is the UID of the input/output being referenced.

Data volumes are a file-specific feature. For this reason, volumes are setup for file-system based input/output DROPs only, namely the FileDROP and the DirectoryContainer types. Other DROP types can instead pass down their dataURL property via the command-line by using placeholders. Placeholders for input DROP dataURLs take the form of "%iDataURLX", where X starts from 0 and refers to the X-th non-filesystem related input. Likewise, output dataURLs are specified as "%oDataURLX". Alternatively users can refer to the dataURL of a specific input or output as "%iDataURL[X]" and "%oDataURL[X]" respectively, where X is the UID of the input/output being referenced.

Additional volume bindings can be specified via the keyword arguments when creating the DockerApp. The host file/directories must exist at the moment of creating the DockerApp; otherwise it will fail to initialize.

Users

A docker container usually runs as root by default. One of the major drawbacks of this is that the output generated by the containerized application will belong also to the root user of the host system, and not to the user running the DALiuGE framework. This DockerApp avoids to run containers as the root user because of this reason. Two parameters, given at construction time, control this behavior:

- ***user***

  If given indicates the user used to run the container. It is assumed that if a user is indicated, the user already exists in the docker image; otherwise the container will actually fail to start. Its default value is *None*, meaning that the container will run as the root user.

- ***ensureUserAndSwitch***
  If the container is run as the root user, this option indicates whether a non-root user with the same UID of the user running this process should be: a) searched for, b) created if it doesn't exist, and c) used to run the command inside the container. This is achieved by prepending some shell commands to the initial user-specified command, which will run as root first, but that finally perform the switch within the container process. Its default value is *True* if *user* is *None*; *False* otherwise.

Using these two options one can thus control the user that will run the command inside the container.

Communication between containers

Although some containerized applications might run on their own, there are cases where applications need to talk to each other in order to advance (like in the case of client-server applications, or in the case of MPI applications). All containers started in the same host (and therefore, all applications running in them) belong by default to the same network, and therefore are already visible.

Applications needing to communicate with other applications should be able to specify the target's IP in their command-line. Since the IP is not known until containers are created, this specification is done using the %containerIp[oid]% placeholder, with 'oid' being the OID of the target DockerApp.

This need to know other DockerApp's IP imposes a sequential order on the startup of the containers, since one needs to be started in order to learn its IP, which is used to start the second. This is handled gracefully by the DockerApp code, with the condition that *self.handleInterest* is invoked where necessary. See *self.handleInterest* for more information about this mechanism.

TODO

Processes in containers might not always exit by themselves, and the containers might need to be manually stopped. This the case for example of an set of MPI processes, where the master container will run the MPI program and the slave containers will run an SSH daemon, where the SSH daemon will not quit automatically once the master process has ended.

Still, we probably will need to differentiate between a forced quit because of a timeout, and a good quit, and therefore we might impose that processes running in a container must quit themselves after successfully performing their task.

`generate_recompute_data()`

Provides a dictionary containing recompute data. At runtime, recomputing, like repeating and rerunning, by default, only shows success or failure. We anticipate that any further implemented behaviour be done at a lower class. :return: A dictionary containing runtime exclusive recompute values.

`handleInterest(`*drop*`)`

Main mechanism through which a DROP handles its interest in a second DROP it isn't directly related to.

A call to this method should be expected for each DROP this DROP is interested in. The default implementation does nothing, but implementations are free to perform any action, such as subscribing to events or storing information.

At this layer only the handling of such an interest exists. The expression of such interest, and the invocation of this method wherever necessary, is currently left as a responsibility of the entity creating the DROPs. In the case of a Session in a DROPManager for example this step would be performed using deployment-time information contained in the dropspec dictionaries held in the session.

`initialize(`***kwargs*`)`

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.dockerapp.**DockerPath**(*path*)

property path

Alias for field number 0

11.3.10 dlg.apps.simple

Applications used as examples, for testing, or in simple situations

class dlg.apps.simple.**AverageArraysApp**(*oid*, *uid*, ***kwargs*)

A BarrierAppDrop that averages arrays received on input. It requires multiple inputs and writes the generated average vector to all of its outputs. The input arrays are assumed to have the same number of elements and the output array will also have that same number of elements.

Keywords:

method: string <[‘mean’]|’median’>, use mean or median as method.

getInputArrays()

Create the input array from all inputs received. Shape is (<#inputs>, <#elements>), where #elements is the length of the vector received from one input.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**CopyApp**(*oid*, *uid*, ***kwargs*)

A BarrierAppDrop that copies its inputs into its outputs. All inputs are copied into all outputs in the order they were declared in the graph.

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**GenericGatherApp**(*oid*, *uid*, ***kwargs*)

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**GenericNpyGatherApp**(*oid*, *uid*, ***kwargs*)

A BarrierAppDrop that reduces then gathers one or more inputs using cumulative operations. function: string <’sum’|’prod’|’min’|’max’|’add’|’multiply’|’maximum’|’minimum’>.

gather_inputs()

gathers each input drop interpreted as an npy drop

reduce_gather_inputs()

reduces then gathers each input drop interpreted as an npy drop

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**GenericNpyScatterApp**(*oid*, *uid*, ***kwargs*)

An APP that splits an object that has a len attribute into <num_of_copies> parts and returns a numpy array of arrays.

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**GenericScatterApp**(*oid*, *uid*, ***kwargs*)

An APP that splits an object that has a len attribute into <numSplit> parts and returns a numpy array of arrays, where the first axis is of length <numSplit>.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**HelloWorldApp**(*oid*, *uid*, ***kwargs*)

An App that writes 'Hello World!' or 'Hello <greet>!' to all of its outputs.

Keywords: greet: string, [World], whom to greet.

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**ListAppendThrashingApp**(*oid*, *uid*, ***kwargs*)

A BarrierAppDrop that appends random integers to a list N times. It does not require any inputs and writes the generated array to all of its outputs.

Keywords:

size: int, number of array elements

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**NullBarrierApp**(*oid*, *uid*, ***kwargs*)

component_meta = <dlg.meta.dlg_component object>

A BarrierAppDrop that doesn't perform any work

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**PickOne**(*oid*, *uid*, ***kwargs*)

Simple app picking one element at a time. Good for Loops.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

writeData(*value*, *rest*)

Prepare the data and write to all outputs

class dlg.apps.simple.**PythonApp**(*oid*, *uid*, ***kwargs*)

A placeholder BarrierAppDrop that just aids the generation of the palette component

class dlg.apps.simple.**RandomArrayApp**(*oid*, *uid*, ***kwargs*)

A BarrierAppDrop that generates an array of random numbers. It does not require any inputs and writes the generated array to all of its outputs.

Keywords:

integer: bool [True], generate integer array low: float, lower boundary (will be converted to int for integer arrays) high: float, upper boundary (will be converted to int for integer arrays) size: int, number of array elements

initialize(*keep_array=False*, ***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**SimpleBranch**(*oid*, *uid*, ***kwargs*)

Simple branch app that is told the result of its condition

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**SleepAndCopyApp**(*oid*, *uid*, ***kwargs*)

A combination of the SleepApp and the CopyApp. It sleeps, then copies

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**SleepApp**(*oid*, *uid*, ***kwargs*)

A BarrierAppDrop that sleeps the specified amount of time (0 by default)

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.simple.**UrlRetrieveApp**(*oid*, *uid*, ***kwargs*)

An App that retrieves the content of a URL

Keywords: URL: string, URL to retrieve.

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

11.3.11 dlg.apps.socket_listener

Module containing the SocketListenerApp, a simple application that listens for incoming data in a TCP socket.

class dlg.apps.socket_listener.**SocketListenerApp**(*oid*, *uid*, ***kwargs*)

A BarrierAppDROP that listens on a socket for data. The server-side socket expects only one client, and assumes that the client will close the connection after all its data has been sent.

This application expects no input DROPs, and therefore raises an exception whenever one is added. On the output side, one or more outputs can be specified with the restriction that they are not ContainerDROPs so data can be written into them through the framework.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

11.3.12 dlg.apps.scp

class dlg.apps.scp.**ScpApp**(*oid*, *uid*, ***kwargs*)

A BarrierAppDROP that copies the content of its single input onto its single output via SSH's scp protocol.

Because of the nature of the scp protocol, the input and output DROPs of this application must both be filesystem-based; i.e., they must be an instance of FileDROP or of DirectoryContainer.

Depending on the physical location of each DROP (this application, and its input and outputs) this application will copy data FROM another host or TO other host. This application's node must thus coincide with one of the two I/O DROPs.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

11.3.13 dlg.apps.archiving

class dlg.apps.archiving.**ExternalStoreApp**(*oid*, *uid*, ***kwargs*)

An application that takes its input DROP (which must be one, and only one) and creates a copy of it in a completely external store, from the point of view of the DALiuGE framework.

Because this application copies the data to an external location, it also shouldn't contain any output, making it a leaf node of the physical graph where it resides.

run()

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

store(*inputDrop*)

Method implemented by subclasses. It should stores the contents of *inputDrop* into an external store.

class dlg.apps.archiving.**NgasArchivingApp**(*oid*, *uid*, ***kwargs*)

An ExternalStoreApp class that takes its input DROP and archives it in an NGAS server. It currently deals with non-container DROPs only.

The archiving to NGAS occurs through the framework and not by spawning a new NGAS client process. This way we can read the different storage types supported by the framework, and not only filesystem objects.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

store(*inDrop*)

Method implemented by subclasses. It should stores the contents of *inputDrop* into an external store.

11.3.14 dlg.apps.crc

Module containing an example application that calculates a CRC value

`class dlg.apps.crc.CRCApp(oid, uid, **kwargs)`

An BarrierAppDROP that calculates the CRC of the single DROP it consumes. It assumes the DROP being consumed is not a container. This is a simple example of an BarrierAppDROP being implemented, and not something really intended to be used in a production system

`run()`

Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

`class dlg.apps.crc.CRCStreamApp(oid, uid, **kwargs)`

Calculate CRC in the streaming mode i.e. A “streamingConsumer” of its predecessor in the graph

`dataWritten(uid, data)`

Callback invoked when *data* has been written into the DROP with UID *uid* (which is one of the streaming inputs of this AppDROP). By default no action is performed

`dropCompleted(uid, status)`

Callback invoked when the DROP with UID *uid* (which is either a normal or a streaming input of this AppDROP) has moved to the COMPLETED or ERROR state. By default no action is performed.

`initialize(**kwargs)`

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

11.4 dlg.data

This package contains several general-purpose data stores in form of DROPs that we have developed as examples and for real-life use. Most of them are based on the `DataDROP`.

Contents

11.4.1 dlg.data.io

class dlg.data.io.**DataIO**

A class used to read/write data stored in a particular kind of storage in an abstract way. This base class simply declares a number of methods that deriving classes must actually implement to handle different storage mechanisms (e.g., a local filesystem or an NGAS server).

An instance of this class represents a particular piece of data. Thus at construction time users must specify a storage-specific unique identifier for the data that this object handles (e.g., a filename in the case of a DataIO class that works with local filesystem storage, or a host:port/fileId combination in the case of a class that works with an NGAS server).

Once an instance has been created it can be opened via its *open* method indicating an open mode. If opened with *OpenMode.OPEN_READ*, only read operations will be allowed on the instance, and if opened with *OpenMode.OPEN_WRITE* only writing operations will be allowed.

buffer() → Union[memoryview, bytes, bytearray, <MagicMock id='140665426717648'>]

Gets a buffer protocol compatible object of the drop data. This may be a zero-copy view of the data or a copy depending on whether the drop stores data in cpu memory or not.

close(***kwargs*)

Closes the underlying storage where the data represented by this instance is stored, freeing underlying resources.

abstract **delete**()

Deletes the data represented by this DataIO

abstract **exists**() → bool

Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

isOpened()

Returns true if the io is currently opened for read or write.

open(*mode: OpenMode*, ***kwargs*)

Opens the underlying storage where the data represented by this instance is stored. Depending on the value of *mode* subsequent calls to *self.read* or *self.write* will succeed or fail.

read(*count: int*, ***kwargs*)

Reads *count* bytes from the underlying storage.

size(***kwargs*) → int

Returns the current total size of the underlying stored object. If the storage class does not support this it is supposed to return -1.

write(*data*, ***kwargs*) → int

Writes *data* into the storage

class dlg.data.io.**ErrorIO**

An DataIO method that throws exceptions if any of its methods is invoked

class dlg.data.io.**FileIO**(*filename*, ***kwargs*)

A file-based implementation of DataIO

`getFileName()`

Returns the drop filename

`dlg.data.io.IOForURL(url)`

Returns a DataIO instance that handles the given URL for reading. If no suitable DataIO class can be found to handle the URL, *None* is returned.

`class dlg.data.io.MemoryIO(buf: BytesIO, **kwargs)`

A DataIO class that reads/write from/into the BytesIO object given at construction time

`class dlg.data.io.NgasIO(hostname, fileId, port=7777, ngasConnectTimeout=2, ngasTimeout=2, length=-1, mimeType='application/octet-stream')`

A DROP whose data is finally stored into NGAS. Since NGAS doesn't support appending data to existing files, we store all the data temporarily in a file on the local filesystem and then move it to the NGAS destination

`delete()`

Deletes the data represented by this DataIO

`class dlg.data.io.NgasLiteIO(hostname, fileId, port=7777, ngasConnectTimeout=2, ngasTimeout=2, length=-1, mimeType='application/octet-stream')`

An IO class whose data is finally stored into NGAS. It uses the ngaslite module of DALiuGE instead of the full client-side libraries provided by NGAS itself, since they might not be installed everywhere.

The *ngaslite* module doesn't support the STATUS command yet, and because of that this class will throw an error if its *exists* method is invoked.

`exists()` → bool

Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

`class dlg.data.io.NullIO`

A DataIO that stores no data

`class dlg.data.io.OpenMode`

Open Mode for Data Drops

`class dlg.data.io.PlasmaFlightIO(object_id: <MagicMock name='mock.ObjectID' id='140665435375184'>, plasma_path='/tmp/plasma', flight_path: ~typing.Optional[str] = None, expected_size: ~typing.Optional[int] = None, use_staging=False)`

A plasma drop managed by an arrow flight network protocol

`class dlg.data.io.PlasmaIO(object_id: <MagicMock name='mock.ObjectID' id='140665435375184'>, plasma_path='/tmp/plasma', expected_size: ~typing.Optional[int] = None, use_staging=False)`

A shared-memory IO reader/writer implemented using plasma store memory buffers. Note: not compatible with PlasmaClient put()/get() which performs data pickling before writing.

`class dlg.data.io.SharedMemoryIO(uid, session_id, **kwargs)`

A DataIO class that writes to a shared memory buffer

11.4.2 dlg.data.drops.data_base

`class dlg.data.drops.data_base.DataDROP(oid, uid, **kwargs)`

A DataDROP is a DROP that stores data for writing with an AppDROP, or reading with one or more AppDROPs.

DataDROPs have two different modes: "normal" and "streaming". Normal DataDROPs will wait until the COMPLETED state before being available as input to an AppDROP, while streaming AppDROPs may be read simutaneously with writing by chunking drop bytes together.

This class contains two methods that need to be overrwritten: *getIO*, invoked by AppDROPs when reading or writing to a drop, and *dataURL*, a getter for a data URI uncluding protocol and address parsed by function *IOForURL*.

`property checksum`

The checksum value for the data represented by this DROP. Its value is automatically calculated if the data was actually written through this DROP (using the *self.write()* method directly or indirectly). In the case that the data has been externally written, the checksum can be set externally after the DROP has been moved to COMPLETED or beyond.

See
self.checksumType

`property checksumType`

The algorithm used to compute this DROP's data checksum. Its value if automatically set if the data was actually written through this DROP (using the *self.write()* method directly or indirectly). In the case that the data has been externally written, the checksum type can be set externally after the DROP has been moved to COMPLETED or beyond.

See
self.checksum

`close(descriptor, **kwargs)`

Closes the given DROP descriptor, decreasing the DROP's internal reference count and releasing the underlying resources associated to the descriptor.

`abstract property dataURL: str`

A URL that points to the data referenced by this DROP. Different DROP implementations will use different URI schemes.

`decrRefCount()`

Decrements the reference count of this DROP by one atomically.

`delete()`

Deletes the data represented by this DROP.

`exists()`

Returns *True* if the data represented by this DROP exists indeed in the underlying storage mechanism

`abstract getIO()` → *DataIO*

Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

`incrRefCount()`

Increments the reference count of this DROP by one atomically.

`isBeingRead()`

Returns *True* if the DROP is currently being read; *False* otherwise `open`(***kwargs*)

Opens the DROP for reading, and returns a "DROP descriptor" that must be used when invoking the read() and close() methods. DROPs maintain a internal reference count based on the number of times they are opened for reading; because of that after a successful call to this method the corresponding close() method must eventually be invoked. Failing to do so will result in DROPs not expiring and getting deleted.

`read`(*descriptor*, *count=65536*, ***kwargs*)

Reads *count* bytes from the given DROP *descriptor*.

`write`(*data: Union[bytes, memoryview]*, ***kwargs*)

Writes the given *data* into this DROP. This method is only meant to be called while the DROP is in INITIALIZED or WRITING state; once the DROP is COMPLETE or beyond only reading is allowed. The underlying storage mechanism is responsible for implementing the final writing logic via the *self.writeMeta()* method.

class `dlg.data.drops.data_base.EndDROP`(*oid*, *uid*, ***kwargs*)

A DROP that ends the session when reached

class `dlg.data.drops.data_base.NullDROP`(*oid*, *uid*, ***kwargs*)

A DROP that doesn't store any data.

property dataURL: str

A URL that points to the data referenced by this DROP. Different DROP implementations will use different URI schemes.

`getIO`()

Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

class `dlg.data.drops.data_base.PathBasedDrop`

Base class for data drops that handle paths (i.e., file and directory drops)

`get_dir`(*dirname*)

dirname will be based on the current working directory If we have a session, it goes into the path as well (most times we should have a session BTW, we should expect *not* to have one only during testing)

Parameters
dirname – str

:returns dir

11.4.3 dlg.data.drops.directorycontainer

class `dlg.data.drops.directorycontainer.DirectoryContainer`(*oid*, *uid*, ***kwargs*)

A ContainerDROP that represents a filesystem directory. It only allows FileDROPs and DirectoryContainers to be added as children. Children can only be added if they are placed directly within the directory represented by this DirectoryContainer.

`delete`()

Deletes the data represented by this DROP.

`exists`()

Returns *True* if the data represented by this DROP exists indeed in the underlying storage mechanism

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

11.4.4 dlg.data.drops.file

class dlg.data.drops.file.**FileDROP**(**args*, ***kwargs*)

A DROP that points to data stored in a mounted filesystem.

Users can fix both the path and the name of a FileDrop using the *filepath* parameter for each FileDrop. We distinguish four cases and their combinations.

1) If not specified the filename will be generated.
2) If it has a '/' at the end it will be treated as a directory name and the filename will the generated.
3) If it does not end with a '/' and it is not an existing directory, it is treated as dirname plus filename.
4) If filepath points to an existing directory, the filename will be generated

In all cases above, if *filepath* does not start with '/' (relative path) then the session directory will be pre-pended to make the path absolute.

property dataURL: str

A URL that points to the data referenced by this DROP. Different DROP implementations will use different URI schemes.

delete()

Deletes the data represented by this DROP.

generate_reproduce_data()

Provides a list of Reproducibility data (specifically). The default behaviour is to return nothing. Per-class behaviour is to be achieved by overriding this method. :return: A dictionary containing runtime exclusive reproducibility data.

getIO()

Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

initialize(***kwargs*)

FileDROP-specific initialization.

sanitize_paths(*filepath: str*) → Union[None, str]

Expand ENV_VARS, but also deal with relative and absolute paths. filepath can be either just be a directory, a directory including a file name, only a directory (both relative and absolute), or just a file name.

Parameters
filepath – string, path and or directory

:returns filepath

setCompleted()

Override this method in order to get the size of the drop set once it is completed.

11.4.5 dlg.data.drops.memory

class dlg.data.drops.memory.**InMemoryDROP**(**args*, ***kwargs*)

A DROP that points data stored in memory.

property dataURL: str

A URL that points to the data referenced by this DROP. Different DROP implementations will use different URI schemes.

generate_reproduce_data()

Provides a list of Reproducibility data (specifically). The default behaviour is to return nothing. Per-class behaviour is to be achieved by overriding this method. :return: A dictionary containing runtime exclusive reproducibility data.

getIO()

Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

class dlg.data.drops.memory.**SharedMemoryDROP**(*oid*, *uid*, ***kwargs*)

A DROP that points to data stored in shared memory. This drop is functionality equivalent to an InMemory drop running in a concurrent environment. In this case however, the requirement for shared memory is explicit.

@WARNING Currently implemented as writing to shmem and there is no backup behaviour.

property dataURL: str

A URL that points to the data referenced by this DROP. Different DROP implementations will use different URI schemes.

getIO()

Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

11.4.6 dlg.data.drops.ngas

class dlg.data.drops.ngas.**NgasDROP**(*oid*, *uid*, ***kwargs*)

A DROP that points to data stored in an NGAS server

property dataURL: str

A URL that points to the data referenced by this DROP. Different DROP implementations will use different URI schemes.

generate_reproduce_data()

Provides a list of Reproducibility data (specifically). The default behaviour is to return nothing. Per-class behaviour is to be achieved by overriding this method. :return: A dictionary containing runtime exclusive reproducibility data.

getIO()

Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

setCompleted()

Override this method in order to get the size of the drop set once it is completed.

11.4.7 dlg.data.drops.plasma

class dlg.data.drops.plasma.**PlasmaDROP**(*oid*, *uid*, ***kwargs*)

A DROP that points to data stored in a Plasma Store

property dataURL: str

A URL that points to the data referenced by this DROP. Different DROP implementations will use different URI schemes.

getIO()

Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

class dlg.data.drops.plasma.**PlasmaFlightDROP**(*oid*, *uid*, ***kwargs*)

A DROP that points to data stored in a Plasma Store

property dataURL: str

A URL that points to the data referenced by this DROP. Different DROP implementations will use different URI schemes.

getIO()

Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

11.4.8 dlg.data.drops.rdbms

class dlg.data.drops.rdbms.**RDBMSDrop**(*oid*, *uid*, ***kwargs*)

A Drop that stores data in a table of a relational database

property dataURL: str

A URL that points to the data referenced by this DROP. Different DROP implementations will use different URI schemes.

generate_reproduce_data()

Provides a list of Reproducibility data (specifically). The default behaviour is to return nothing. Per-class behaviour is to be achieved by overriding this method. :return: A dictionary containing runtime exclusive reproducibility data.

getIO()

Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

initialize(***kwargs*)

Performs any specific subclass initialization.

kwargs contains all the keyword arguments given at construction time, except those used by the constructor itself. Implementations of this method should make sure that arguments in the *kwargs* dictionary are removed once they are interpreted so they are not interpreted by accident by another method implementations that might reside in the call hierarchy (in the case that a subclass implementation calls the parent method implementation, which is usually the case).

insert(*vals: dict*)

Inserts the values contained in the `vals` dictionary into the underlying table. The keys of `vals` are used as the column names.

select(*columns=None*, *condition=None*, *vals=()*)

Returns the selected values from the table. Users can constrain the result set by specifying a list of `columns` to be returned (otherwise all table columns are returned) and a `condition` to be applied, in which case a list of `vals` to be applied as query parameters can also be given.

11.4.9 dlg.data.drops.json_drop

A DROP for a JSON file

11.4.10 dlg.data.drops.s3_drop

Drops that interact with S3

11.5 dlg.dropmake

Implementation of a DataFlow Manager based on the original SKA SDP architecture.

Contents

11.5.1 dlg.dropmake.pg_generator

The DALiuGE resource manager uses the requested logical graphs, the available resources and the profiling information and turns it into the partitioned physical graph, which will then be deployed and monitored by the Physical Graph Manager

`dlg.dropmake.pg_generator.`**`fill`**(*lg*, *params*)

Logical Graph + params -> Filled Logical Graph

`dlg.dropmake.pg_generator.`**`partition`**(*pgt*, *algo*, *num_partitions=1*, *num_islands=1*, *partition_label='partition'*, *show_gojs=False*, ***algo_params*)

Partitions a Physical Graph Template

`dlg.dropmake.pg_generator.`**`resource_map`**(*pgt*, *nodes*, *num_islands=1*, *co_host_dim=True*)

Maps a Physical Graph Template *pgt* to *nodes*

`dlg.dropmake.pg_generator.`**`unroll`**(*lg*, *oid_prefix=None*, *zerorun=False*, *app=None*)

Unrolls a logical graph

11.5.2 dlg.dropmake.scheduler

`class` `dlg.dropmake.scheduler.`**`DAGUtil`**

Helper functions dealing with DAG

`static build_dag_from_drops`(*drop_list*, *embed_drop=True*, *fake_super_root=False*)

return a networkx Digraph (DAG) :param: fake_super_root whether to create a fake super root node in the DAG If set to True, it enables edge zero-based scheduling agorithms to make more aggressive merging

static ganttchart_matrix(*G*, *topo_sort=None*)

Return a M (# of DROPs) by N (longest path length) matrix

static get_longest_path(*G*, *weight='weight'*, *default_weight=1*, *show_path=True*, *topo_sort=None*)

Ported from: https://github.com/networkx/networkx/blob/master/networkx/algorithms/dag.py Added node weight

Returns the longest path in a DAG If G has edges with 'weight' attribute the edge data are used as weight values. :param: G Graph (NetworkX DiGraph) :param: weight Edge data key to use for weight (string) :param: default_weight The weight of edges that do not have a weight attribute (integer) :return: a tuple with two elements: *path* (list), the longest path, and *path_length* (float) the length of the longest path.

static get_max_antichains(*G*)

return a list of antichains with Top-2 lengths

static get_max_dop(*G*)

Get the maximum degree of parallelism of this DAG return : int

static get_max_width(*G*, *weight='weight'*, *default_weight=1*)

Get the antichain with the maximum "weighted" width of this DAG weight: float (for example, it could be RAM consumption in GB) Return : float

static label_schedule(*G*, *weight='weight'*, *topo_sort=None*)

for each node, label its start and end time

static metis_part(*G*, *num_partitions*)

Use metis binary executable (instead of library) This is used only for testing when libmetis halts unexpectedly

static prune_antichains(*antichains*)

Prune a list of antichains to keep those with Top-2 lengths antichains is a Generator (not a list!)

class dlg.dropmake.scheduler.**KFamilyPartition**(*gid*, *max_dop*, *global_dag=None*)

A special case (K = 1) of the Maximum Weighted K-families based on the Theorem 3.1 in http://fmdb.cs.ucla.edu/Treports/930014.pdf

add_node(*u*)

Add a single node u to the partition

class dlg.dropmake.scheduler.**MinNumPartsScheduler**(*drop_list*, *deadline*, *max_dop=8*, *dag=None*, *optimistic_factor=0.5*)

A special type of partition that aims to schedule the DAG on time but at minimum cost. In this particular case, the cost is the number of partitions that will be generated. The assumption is # of partitions (with certain DoP) more or less represents resource footprint.

is_time_critical(*u*, *uw*, *unew*, *v*, *vw*, *vnew*, *curr_lpl*, *ow*, *rem_el*)

This is called ONLY IF either can_add on partition has returned "False" or the new critical path is longer than the old one at each iteration

Parameters:
u - node u, v - node v, uw - weight of node u, vw - weight of node v curr_lpl - current longest path length, ow - current edge weight rem_el - remainig edges to be zeroed ow - original edge length

Returns:
Boolean

It looks ahead to compute the probability of time being critical and compares that with the _optimistic_factor probility = (num of edges need to be zeroed to meet the deadline) / (num of remaining unzeroed edges) `override_cannot_add()`

Whether this scheduler will override the False result from *Partition.can_add()*

`class dlg.dropmake.scheduler.MySarkarScheduler(`*drop_list*, *max_dop=8*, *dag=None*, *dump_progress=False*`)`

Based on "V. Sarkar, Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors. Cambridge, MA: MIT Press, 1989."

Main change We do not order independent tasks within the same cluster. This could blow the cluster, therefore we allow for a cost constraint on the number of concurrent tasks (e.g. # of cores) within each cluster

Why 1. we only need to topologically sort the DAG once since we do not add new edges in the cluster 2. closer to requirements 3. adjustable for local schedulers

Similar ideas: http://stackoverflow.com/questions/3974731

`is_time_critical(`*u*, *uw*, *unew*, *v*, *vw*, *vnew*, *curr_lpl*, *ow*, *rem_el*`)`

Returns
True

MySarkarScheduler always returns False

`override_cannot_add()`

Whether this scheduler will override the False result from *Partition.can_add()*

`partition_dag()`

Return a tuple of

1. the # of partitions formed (int)
2. the parallel time (longest path, int)
3. partition time (seconds, float)

`reduce_partitions(`*parts*, *g_dict*, *G*`)`

further reduce the number of partitions by merging partitions whose max_dop is less than capacity

step 1 - sort partition list based on their
_max_dop of num_cpus as default

step 2 - enumerate each partition p to see merging
between p and its neighbour is feasible

`class dlg.dropmake.scheduler.PSOScheduler(`*drop_list*, *max_dop=8*, *dag=None*, *deadline=None*, *topk=30*, *swarm_size=40*`)`

Use the Particle Swarm Optimisation to guide the Sarkar algorithm https://en.wikipedia.org/wiki/Particle_swarm_optimization

The idea is to let "edgezeroing" becomes the search variable X The number of dimensions of X is the number of edges in DAG Possible values for each dimension is a discrete set {1, 2, 3} where: * 10 - no zero (2 in base10) + 1 * 00 - zero w/o linearisation (0 in base10) + 1 * 01 - zero with linearisation (1 in base10) + 1

if (deadline is present):

the objective function sets up a partition scheme such that

(1) DoP constrints for each partiiton are satisfied based on X[i] value, reject or linearisation

(2) returns num_of_partitions

constrain function:

1. makespan < deadline

else:

the objective function sets up a partition scheme such that

(1) DoP constrints for each partiiton are satisfied based on X[i] value, reject or linearisation

(2) returns makespan

`constrain_func`(*x*)

Deadline - critical_path >= 0

`objective_func`(*x*)

x is a list of values, each taking one of the 3 integers: 0,1,2 for an edge indices of x is identical to the indices in G.edges().sort(key='weight')

`partition_dag`()

Returns a tuple of:

1. the # of partitions formed (int)
2. the parallel time (longest path, int)
3. partition time (seconds, float)
4. a list of partitions (Partition)

class dlg.dropmake.scheduler.**Partition**(*gid*, *max_dop*)

Logical partition, multiple (1 ~ N) of these can be placed onto a single physical resource unit

Logical partition can be nested, and it somewhat resembles the *dlg.manager.drop_manager*

`add`(*u*, *v*, *gu*, *gv*, *sequential=False*, *global_dag=None*)

Add nodes u and/or v to the partition if sequential is True, break antichains to sequential chains

`add_node`(*u*, *weight*)

Add a single node u to the partition

`can_add`(*u*, *v*, *gu*, *gv*)

Check if nodes u and/or v can join this partition A node may be rejected due to reasons such as: DoP overflow or completion time deadline overdue, etc.

`probe_max_dop`(*u*, *v*, *unew*, *vnew*, *update=False*)

An incremental antichain (which appears significantly more efficient than the networkx antichains) But only works for DoP, not for weighted width

`remove`(*n*)

Remove node n from the partition

`property schedule`

Get the schedule assocaited with this partition

class dlg.dropmake.scheduler.**Schedule**(*dag*, *max_dop*)

The scheduling solution with schedule-related properties

`property efficiency`

resource usage percentage (integer)

property schedule_matrix

Return: a self._lpl x self._max_dop matrix
(X - time, Y - resource unit / parallel lane)

property workload

Return: (integer)
the mean # of resource units per time unit consumed by the graph/partition

class dlg.dropmake.scheduler.**Scheduler**(*drop_list*, *max_dop=8*, *dag=None*)

Static Scheduling consists of three steps: 1. partition the DAG into an optimal number (M) of partitions goal - minimising execution time while maintaining intra-partition DoP 2. merge partitions into a given number (N) of partitions (if M > N) goal - minimise logical communication cost while maintaining load balancing 3. map each merged partition to a resource unit goal - minimise physical communication cost amongst resource units

map_partitions()

map logical partitions to physical resources

merge_partitions(*num_partitions*, *bal_cond=1*)

Merge M partitions into N partitions where N < M
implemented using METIS for now

bal_cond: load balance condition (integer):
0 - workload, 1 - CPU count (faster to evaluate than workload)

exception dlg.dropmake.scheduler.**SchedulerException**

11.5.3 dlg.dropmake.pg_manager

Refer to https://confluence.ska-sdp.org/display/PRODUCTTREE/C.1.2.4.4.4+DFM+Physical+Graph+Manager

class dlg.dropmake.pg_manager.**PGManager**(*root_dir*)

Physical Graph Manager

add_pgt(*pgt*, *lg_name*)

Dummy impl. using file system for now (thread safe) TODO - use proper graph databases to manage all PGTs

Return:
A unique PGT id (handle)

get_gantt_chart(*pgt_id*, *json_str=True*)

Return:
the gantt chart matrix (numarray) given a PGT id

get_pgt(*pgt_id*)

Return:
The PGT object given its PGT id

get_schedule_matrices(*pgt_id*, *json_str=True*)

Return:
a list of schedule matrices (numarrays) given a PGT id

`class dlg.dropmake.pg_manager.PGUtil`

Helper functions dealing with Physical Graphs

`static vstack_mat`(*A*, *B*, *separator=False*)

Vertically stack two matrices that may have different # of colums

Param
A matrix A (2d numpy array)

Param
B matrix B (2d numy array)

Param
separator whether to add an empty row separator between the two matrices (boolean)

Returns
the vertically stacked matrix (2d numpy array)

CHAPTER

TWELVE

CITATIONS

As you use DALiuGE for your exciting projects, please cite the following paper:

Wu, C., Tobar, R., Vinsen, K., Wicenec, A., Pallot, D., Lao, B., Wang, R., An, T., Boulton, M., Cooper, I. and Dodson, R., 2017. DALiuGE: A Graph Execution Framework for Harnessing the Astronomical Data Deluge. Astronomy and Computing, 20, pp.1-15. (2017)

PYTHON MODULE INDEX

INDEX