daliuge Documentation

Release 0.2.0

ICRAR

Sep 04, 2018

Contents

1 Introduction **3**

2 Installation **5**
- 2.1 Requirements . . . 5
- 2.2 Installing . . . 5
- 2.3 Docker images . . . 6

3 Overview **7**
- 3.1 Concepts and Background . . . 7
- 3.2 DROPs . . . 9
- 3.3 Graphs . . . 11
- 3.4 DROP Managers . . . 15
- 3.5 Data Lifecycle Manager . . . 17
- 3.6 References . . . 18

4 Graph development **19**
- 4.1 Using the Logical Graph Editor . . . 19
- 4.2 Directly creating a Physical Graph . . . 19
- 4.3 Using `dlg.delayed()` . . . 20

5 Application development **21**
- 5.1 Class . . . 21
- 5.2 I/O . . . 21

6 Using the Logical Graph Editor **23**
- 6.1 General . . . 23
- 6.2 Components . . . 24
- 6.3 Examples . . . 24

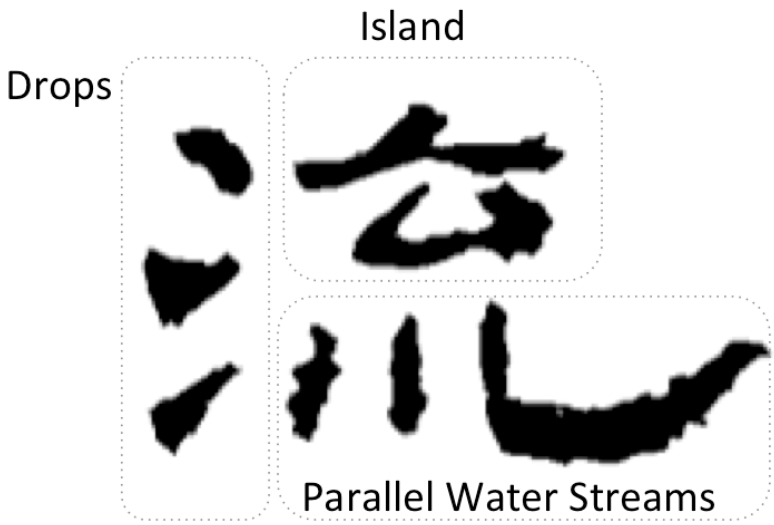
7 API **27**
- 7.1 dlg . . . 27
- 7.2 dlg.manager . . . 42
- 7.3 dlg.apps . . . 47
- 7.4 dlg.dropmake . . . 54

8 Citations **65**

Python Module Index **67**

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.



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 1

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 has 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.

In *Overview* 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 2

Installation

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:

- boost-python
- boost-system
- boost-devel
- gcc >= 4.8

2.2 Installing

DALiuGE is based on setuptools, and thus it follows the standard python installation procedures. The preferred way of installing the latest stable version of DALiuGE is by using `pip`:

```
pip install --process-dependency-links daliuge
```

If you want to build from the latest sources you can get them from here:

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

If a system-wide installation is required, then the following commands can be issued:

```
sudo pip --process-dependency-links install .
```

If `pip` is not available, you can also use a different approach with:

```
python setup.py build
sudo python setup.py install
```

If a virtualenv is loaded, then DALiuGE can be installed on it by simply running:

```
pip install --process-dependency-links .
```

Again, if `pip` is not available, you can use the simpler form:

```
python setup.py install
```

There is a known issue in some systems when installing the `python-daemon` dependency, which **needs** to be installed via `pip`.

2.3 Docker images

Docker images can be built using the Dockerfiles under the `docker` directory. Please refer to the `README` file in the docker directory for more information.

CHAPTER 3

Overview

The following sections give an overview of the different modules present in DALiuGE.

3.1 Concepts and Background

This section briefly introduces key concepts and motivations underpinning DALiuGE.

3.1.1 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.

3.1.2 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.

3.1.3 Data-driven

In developing the DALiuGE prototype, we have extended the "traditional" dataflow model by integrating data lifecycle management, graph execution engine, and cost-optimal resource allocation into a coherent data-driven 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, are 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-Driven 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 computation 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).

3.1.4 DALiuGE Functions

The DALiuGE prototype provides eight Graph-based functions as shown in Fig. 3.1.

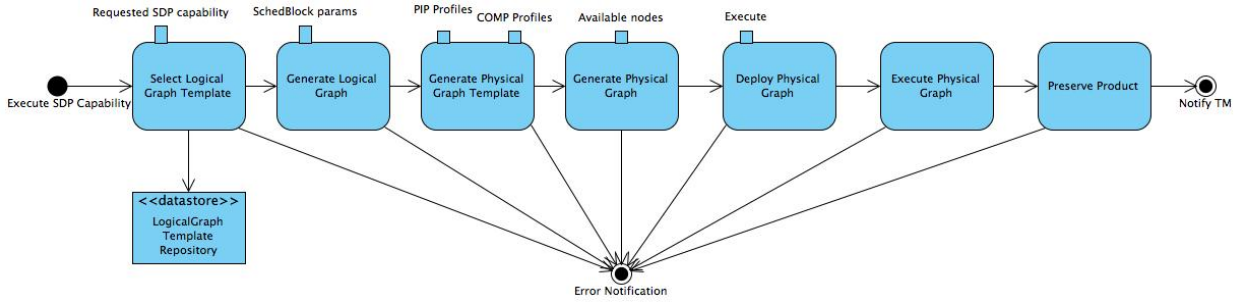


Fig. 3.1: Graph-based Functions of the DALiuGE Prototype

The *Graphs* section will go through implementation details for each function. Here we briefly discuss how they work together in our data-driven framework.

- First of all, the *Logical Graph Template* (topleft in Fig. 3.1) represents high-level data processing capabilities. In the case of SDP, they could be, for example, "Process Visibility Data" or "Stage Data Products".
- All logical graph templates are managed by the *LogicalGraph Template Repository* (bottomleft in Fig. 3.1). The logical graph template is first selected from this repository for a specific pipeline and is then filled with scheduling block parameters. This generates a *Logical Graph*, expressing a pipeline with resource-oblivious dataflow constructs.
- Using profiling information of pipeline components and COMP hardware resources, the DALiuGE prototype 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.
- Before an observation starts, DALiuGE deploys all the DROPs onto these resources as per the location information stated in the physical graph. The deployment process is facilitated through *DROP Managers*, which are daemon processes managing deployed DROPs on designated resources.
- 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.

3.2 DROPs

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

3.2.1 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.

3.2.2 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.

3.2.3 Relationships

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

- 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).
- 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.

- 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).
- 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.

3.2.4 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 DROPs with their associated I/O storage classes, including in-memory, file-base and S3 storages.

3.2.5 DROP Channels

DROPs that are connected by an edge in a physical graph but are deployed on separate nodes or islands from each other are automatically given a Pyro stub (remote method invocation interface) to allow them to communicate with each other. It's the job of the Master DROP and Island Managers to generate and exchange stubs between DROP instances before the graph is deployed to the various data islands and nodes within islands respectively. 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, as a result, basic method calls are used between DROP instances.

3.2.6 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:

- Portability.
- Versioning and component reuse.
- Lightweight footprint.
- 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.drop* 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.

3.3 Graphs

A processing pipeline in DALiuGE is described by a Directed Graph where the nodes denote both task (application DROPs) and data (data DROPs). The edges denote execution dependencies between DROPs. Section *DALiuGE Functions* has briefly introduced graph-based functions in DALiuGE. This section provides implementation details in the DALiuGE prototype.

3.3.1 Logical Graph

A logical graph is a compact representation of the logical operations in a processing pipeline without concerning underlying hardware resources. Such operations are referred to as *construct* in a logical graph. The relationship between a DROP and a construct resembles the one between and object and a class in Object Oriented programming languages. In other words, most constructs are DROP templates and multiple DROPs correspond to a single construct.

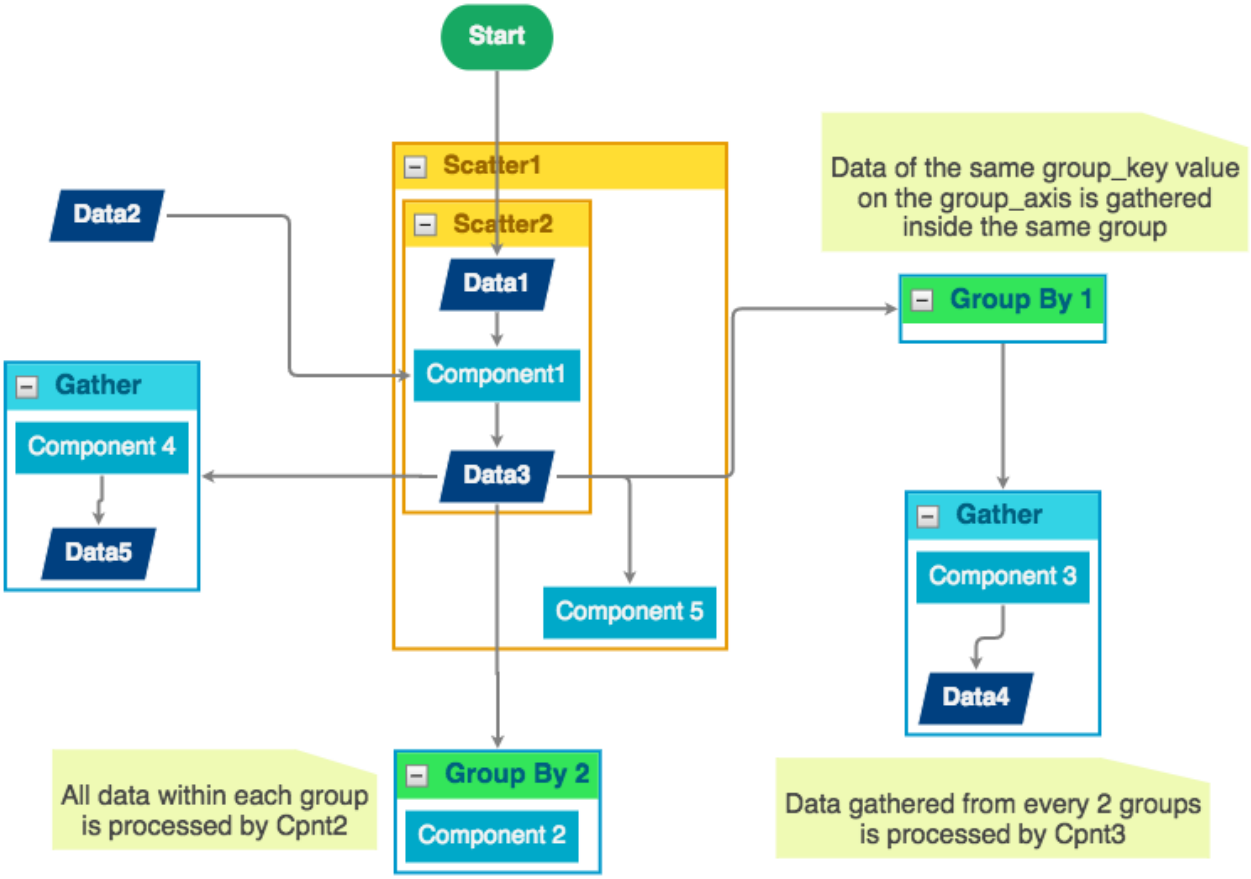


Fig. 3.2: An example of a logical graph with data constructs (e.g. Data1 - Data5), component constructs (i.e. Component1 - Component5), and control flow constructs (Scatter, Gather, and Group-By). This example can be viewed online in the DALiuGE prototype.

Construct properties

Each construct has several associated properties that users have control over during the development of a logical graph. For Component and Data constructs the **Execution time** and **Data volume** are two very important properties. Such

properties can be directly obtained from parametric models or estimated from the profiling information (e.g. pipeline component workload characterisation) and COMP platform specification.

Control flow constructs

Control flow 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. Constructs inside a *Scatter* construct represent a group of components consuming a single data partition within the enclosing *Scatter*. A useful property of *Scatter* is `num_of_copies`. In the example in Fig. 3.2, if the `num_of_copies` 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 `num_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. 3.2), in which case, data held in a sequence of groups are processed together by components enclosed by *Gather*.
- **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. The current DALiuGE prototype 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.
- **Loop** indicates iterations. Constructs inside a *Loop* represent a group of components and data that will be repeatedly executed / produced for a fixed number of times. Given the basic DROP principle of "writing once, read many times", the current DALiuGE prototype does not support dynamic branch condition for *Loop*. Instead, each *Loop* construct has a property named `num_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 `num_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. 3.3.

Repository

The DALiuGE prototype uses a Web-based logical graph editor as the default user interface to the underlying *logical graph repository*, which currently is simply a managed POSIX file system directory. Each logical graph is physically stored as a JSON-formatted textual file, and can be accessed and modified remotely through the logical graph editor via the RESTful interface. For example, the JSON file for the continuous imaging pipeline as shown partially in Fig. 3.3 can be accessed through HTTP GET. The editor also provides a Web-based JSON editor so that users can directly change the graph JSON content inside the repository.

Select template

While the DALiuGE logical graph editor does not differentiate between logical graph and *logical graph template*, users can create either of them using the editor (after all, the only differences between these two are the populated values for some parameters). Once a template is created or selected, users can simply copy and paste the JSON content into the new logical graph and fill in those parameter values (as construct properties) using the editor. Note that the public version of the logical graph editor has not yet opened its "create new logical graph" API.

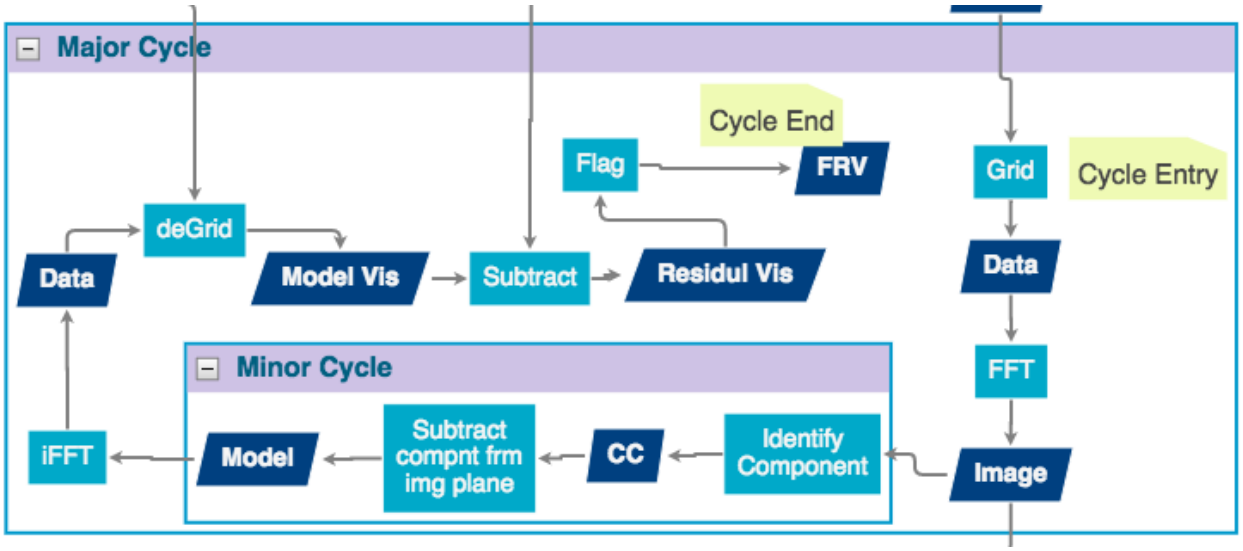


Fig. 3.3: A nested-Loop (minor and major cycle) example of logical graph for a continuous imaging pipeline. This example can be viewed online in the DALiuGE prototype.

3.3.2 Translation

While a logical graph provides a compact way to express complex processing logic, it contains high level control flow specifications that 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 essentially creates all DROPs and is implemented in the *dlg.dropmake* module.

Basic steps

DropMake in the DALiuGE prototype 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* construct as shown in Fig. 3.2. Any 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. *DropIsland*) 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 COMP platform or the Local Monitor & Control (LMC). 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**.

Under the assumption of uniform resources (e.g. each node has identical capabilities), graph partitioning is equivalent to resource mapping since mapping involves simple round-robin all available resources. In this case, graph partitioning algorithms (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 has not yet supported, 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*.

Algorithms

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. The DALiuGE prototype 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 made available shortly. With these algorithms, the DALiuGE prototype currently attempts to address the following translation problems:

- **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* DropIslands (*M* <= *N*) from the *physical graph template* such that (1) the total volume of data traveling between two distinct DropIslands is minimised, and (2) the workload variations measured in aggregated **execution time** (DROP property) between a pair of DropIslands 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 DropIsland 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 DropIslands 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 DropIsland 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 DropIslands** but subject to (1) a given **completion time deadline**, and (2) a given *DoP* (e.g. number of cores per node) that each DropIsland 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 pipeline within, say, 5 minutes, the schedule that consumes less resources is preferred. Since a DropIsland is mapped onto resources, and its capacity is already constrained by a given DoP, the number of DropIslands is proportional to the amount of resources needed. Consequently, schedules that require less number of DropIslands 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.

3.3.3 Physical Graph

The *Translation* process produces the *physical graph specification*, which, once deployed and instantiated "live", becomes the physical graph, a collection of inter-connected DROPs in a distributed execution plan across multiple resource units. The nodes of a physical graph are DROPs representing either data or applications. The two DROP nodes

connected by an edge always have different types from each other. This establishes a set of reciprocal relationships between DROPs:

- A data DROP is the *input* of an application DROP; on the other hand the application is a *consumer* of the data DROP.
- Likewise, 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.
- Finally, a data DROP can be the *output* of an application DROP, in which case the application is the *producer* of the data DROP.

Physical graph specifications are the final (and only) graph products that will be submitted to the *DROP Managers*. Once DROP managers accept a physical graph specification, it is their responsibility to create and deploy DROP instances on their managed resources as prescribed in the physical graph specification such as partitioning information (produced during the *Translation*) that allows different managers to distribute graph partitions (i.e. DropIslands) across different nodes and Data Islands by setting up proper *DROP Channels*. The fact that physical graphs are made of DROPs means that they describe exactly what an *Execution* consists of. In this sense, the physical graph is the graph execution engine.

In addition to DROP managers, the DALiuGE prototype also includes a *Physical Graph Manager*, which allows users to manage all currently running and past physical graphs within the system. Although the current *Physical Graph Manager* implementation only supports to "add" and "get" physical graph specifications, features such as graph event monitoring (through the DROP *Events* subscription mechanism) and the graph statistics dashboard will be added in the near future.

3.3.4 Execution

A physical graph has the ability to advance its own execution. This is internally implemented via the DROP event mechanism as follows:

- Once a data DROP moves to the COMPLETED state it will fire an event to all its consumers. Consumers (applications) will then deem 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.
- On the other hand, data DROPs receive an even 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.

3.4 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:

- A *Node DROP Manager* exists for every node in the system.
- Nodes are grouped into *Data Islands*, and thus a *Data Island DROP Manager* exists at the Data Island level.
- On top of the Data Islands is the *Master DROP Manager*.

3.4.1 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:

- Submitting the same physical graph to a DROP Manager will create two different sessions
- 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.

3.4.2 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 Pyro to allow remote method executions on them.

3.4.3 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 Pyro proxies and linking them correspondingly.

3.4.4 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.

3.4.5 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>
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.

3.4.6 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.

3.5 Data Lifecycle Manager

As mentioned in *Introduction* and *Data-driven* 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) prototype. 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 the DALiuGE prototype are:

- Automatically expire Drops; i.e., moves them from the **COMPLETED** state into the **EXPIRED** state, after which they are not readable anymore.
- Automatically delete data from Drops in the **EXPIRED** state, and move the Drops into the **DELETED** state.
- 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:

- 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.
- 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.

3.6 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

CHAPTER 4

Graph development

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

As explained in *Graphs*, DALiuGE describes computations in terms of Directed Acyclic Graphs. Two different types of graphs are used throughout application development: *Logical Graphs*, a high-level, compact representation of the application logic, and *Physical Graphs*, a detailed description of each individual processing step. When submitting a graph for execution, users submit physical graphs to the runtime component of DALiuGE. 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*.

Given all the above, the following graph development techniques are available for users to creates graphs and submit them for execution:

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

4.1 Using the Logical Graph Editor

The *Using the Logical Graph Editor* section has some examples on how to use the Logical Graph Editor to compose Logical Graphs.

Please be aware that this section is old and incomplete, and also refers to the "old" Logical Graph Editor, which although hasn't been removed, will soon be deprecated. A new Logical Graph Editor called *EAGLE* is currently in the works. This new editor implements more features, is more complete in its support, and is visually easier to follow. More information about it will come soon.

4.2 Directly creating a Physical Graph

In some cases using the Logical Graph Editor is not possible. This can happen because its (currently) limited capabilities, or because somehow some information would be lost during the translation process.

In these cases, producing the Physical Graph directly is still possible. Once a Physical Graph is produced, it can be partitioned and mapped (see the steps in *Translation*) either via the DALiuGE utilities, or by the user directly. Finally, the Physical Graph can be sent to one of the Drop Managers for execution.

4.3 Using `dlg.delayed()`

CHAPTER 5

Application development

This section describes what developers need to do to write a new class that can be used as an Application Drop in DALiuGE.

5.1 Class

Developers need to write a new python class that derives from the *dlg.drop.BarrierAppDROP* class. 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*, that receives no arguments, and executes the logic of the application.

5.2 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.

CHAPTER 6

Using the Logical Graph Editor

- *General*
- *Components*
  - *ShellApp*
  - *Data*
  - *Scatter*
- *Examples*
  - *Simple scatter*

These are some guidelines on how to use the Logical Graph Editor included in DALiuGE.

6.1 General

On the left-hand side is the palette where different component types are shown. Users can drag these components and drop them in the central area where a Logical Graph will be build up. When hovering the mouse over a component on the Logical Graph different connection points appear on the borders of the component. By clicking on these and dragging the mouse over to a different component users can draw an arrow between two components signaling a relationship between the two.

By clicking on a component's text users can also change the label shown by that component on the Logical Graph. This is useful for readability and has no impact on the final output of the Logical Graph.

Also when clicking on a component the editor will show the component's properties on the bottom-right corner. Users can change here some values associated to the components. Different components support different properties.

6.2 Components

6.2.1 ShellApp

The ShellApp component represents a bash shell command. The command to be run is written using its different Arg properties. For example, to run echo 123 users would have to write echo in Arg01 and 123 in Arg02.

When referring to the application's inputs and outputs in the command line %i[X] and %o[X] can be used, respectively, where X refers to the key property of the referenced input/output. These placeholders will eventually be replaced with the file path of the corresponding inputs or outputs when executing the command. For inputs and outputs that are not filesystem-based the %iDataURL[X] and %oDataURL[X] placeholders can be used instead.

Note that sometimes an application is connected to an input or output component representing more than one physical input or output (e.g., an application outside a Scatter component connected to a Data component inside the Scatter). In these cases the corresponding placeholder will expand to a semicolon-delimited list of paths or URLs. It will be the responsibility of the application to deal with these cases.

6.2.2 Data

The Data component represents a payload. When transitioning from the Logical Graph to the Physical Graph these components generate InMemory drops currently, but support will be added in the future to generate drops with different storage mechanisms.

6.2.3 Scatter

The Scatter component represents parallel branches of execution. Is is represented by a yellow-bordered box that allows other components to be placed within. All the contents of the body of a Scatter will be replicated as parallel branches of execution when performing the Logical Graph to Physical Graph transition depending on the value of its num_of_copies property.

Components *outside* a Scatter can be connected to components *inside* the Scatter. In such cases, when he Physical Graph is generated, the *outside* component will appear as connected to the many copies generated from *inside* the Scatter.

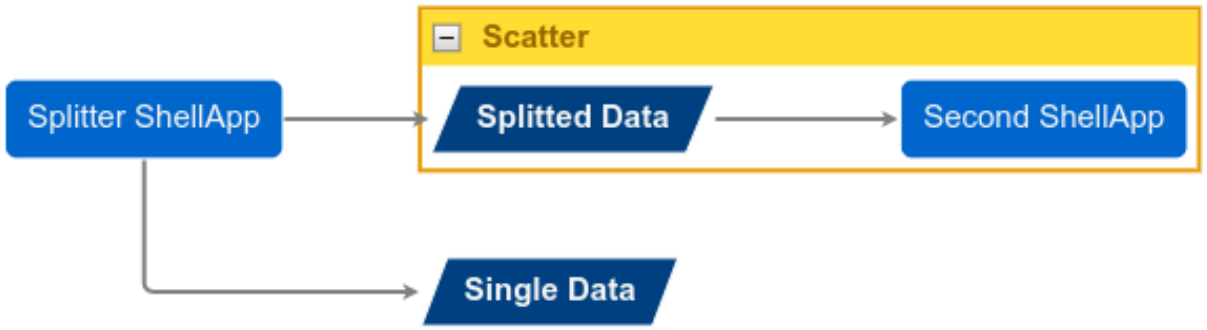
6.3 Examples

6.3.1 Simple scatter

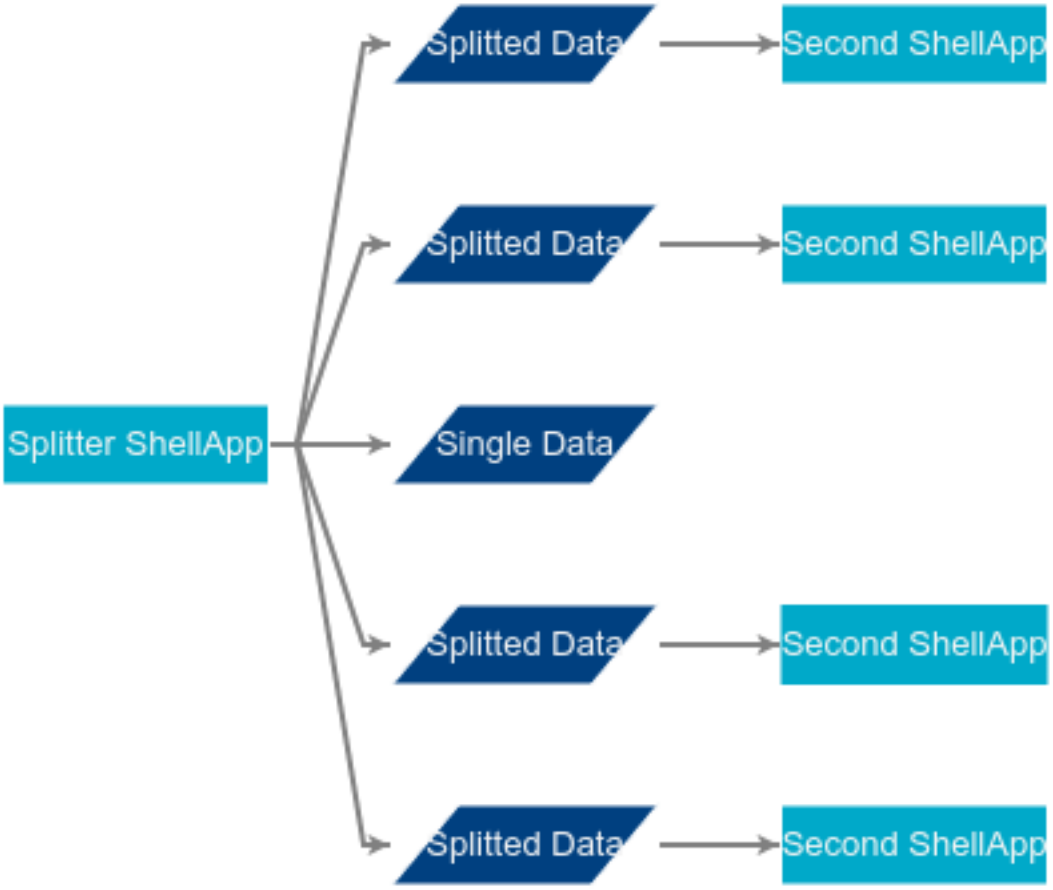
In this example we will have an application producing many outputs, identical in nature, which are then processed in parallel by a second application. The initial application also outputs a different, single file.

To do this we drop a ShellApp into the Logical Graph. Next to it we drop a Scatter component. Inside the Scatter component we drop a Data component, and next to the Data component we finally drop a second ShellApp component. One can then draw an arrow from the first ShellApp to the Data component, and a second one from the Data component to the second ShellApp. Finally, drop a Data component outside the Scatter and draw an arrow from the first ShellApp component into it. Optionally change the names of the components for readability.

The final result should look like this:



If you save the Logical Graph form above, and then generate the Physical Graph it will look like this:



Here it can be clearly seen how the `Scatter` component's body has been replicated according to its configuration. It also shows how the first application now produces multiple outputs.

CHAPTER 7

API

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.

7.1 dlg

This package contains the modules implementing the core functionality of the system.

Contents

- *dlg*
  - *dlg.event*
  - *dlg.io*
  - *dlg.drop*
  - *dlg.s3_drop*
  - *dlg.droputils*
  - *dlg.utils*
  - *dlg.graph_loader*
  - *dlg.delayed*

7.1.1 dlg.event

class dlg.event.**Event**
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*, *eventType=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*, *eventType=None*)
Unsubscribes *listener* from events fired by this object.

7.1.2 dlg.io

class dlg.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.

close(***kwargs*)
Closes the underlying storage where the data represented by this instance is stored, freeing underlying resources.

delete()
Deletes the data represented by this DataIO

exists()
Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

open(*mode*, ***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*, ***kwargs*)
Reads *count* bytes from the underlying storage.

write(*data*, ***kwargs*)
Writes *data* into the storage

class dlg.io.**ErrorIO**
An DataIO method that throws exceptions if any of its methods is invoked

delete()
Deletes the data represented by this DataIO

exists()
Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

class dlg.io.**FileIO**(*filename*, ***kwargs*)

delete()
Deletes the data represented by this DataIO

exists()
Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

dlg.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.io.**MemoryIO**(*buf*, ***kwargs*)
A DataIO class that reads/write from/into the BytesIO object given at construction time

delete()
Deletes the data represented by this DataIO

exists()
Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

class dlg.io.**NgasIO**(*hostname*, *fileId*, *port=7777*, *ngasConnectTimeout=2*, *ngasTimeout=2*, *length=-1*)
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

exists()
Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

class dlg.io.**NgasLiteIO**(*hostname*, *fileId*, *port=7777*, *ngasConnectTimeout=2*, *ngasTimeout=2*, *length=-1*)
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.

delete()
Deletes the data represented by this DataIO

exists()
Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

class dlg.io.**NullIO**
A DataIO that stores no data

delete()
Deletes the data represented by this DataIO

exists()
Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

class dlg.io.**ShoreIO**(*doid*, *column*, *row*, *rows=1*, *address=None*, ***kwargs*)

delete()
Deletes the data represented by this DataIO

exists()
Returns *True* if the data represented by this DataIO exists indeed in the underlying storage mechanism

7.1.3 dlg.drop

Module containing the core DROP classes.

class dlg.drop.**AbstractDROP**(***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(***kwargs*)
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(***kwargs*)
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(***kwargs*)
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.

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*

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(***kwargs*)
Closes the given DROP descriptor, decreasing the DROP's internal reference count and releasing the underlying resources associated to the descriptor.

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

See *self.addConsumer()*

dataURL()
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.

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.

exists()
Returns *True* if the data represented by this DROP exists indeed in the underlying storage mechanism

getIO()
Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

handleEvent(***kwargs*)
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.

incrRefCount()
Increments the reference count of this DROP by one atomically.

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).

isBeingRead()
Returns *True* if the DROP is currently being read; *False* otherwise

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

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.

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.

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.

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

precious
Whether this DROP should be considered as 'precious' or not

producerFinished(***kwargs*)
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.

producers
The list of producers that write to this DROP

See *self.addProducer()*

read(*descriptor*, *count=4096*, ***kwargs*)
Reads *count* bytes from the given DROP *descriptor*.

setCompleted(***kwargs*)
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(***kwargs*)
Moves this DROP to the ERROR state.

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.

status
The current status of this DROP.

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

See *self.addStreamingConsumer()*

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.

write(***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.drop.**AppDROP**(***kwargs*)
An AppDROP is a DROP representing an application that reads data from one or more DROPs (its inputs), and writes data onto one or more DROPs (its outputs).

AppDROPs accept two different kind of inputs: "normal" and "streaming" inputs. Normal inputs are DROPs that must be on the COMPLETED state (and therefore their data must be fully written) before this application is run, while streaming inputs are DROPs that feed chunks of data into this application as the data gets written into them.

This class contains two methods that should be overwritten as needed by subclasses: *dropCompleted*, invoked when input DROPs move to COMPLETED, and *dataWritten*, invoked with the data coming from streaming inputs.

How and when applications are executed is completely up to the user, 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*.

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.

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).

inputs
The list of inputs set into this AppDROP

outputs
The list of outputs set into this AppDROP

streamingInputs
The list of streaming inputs set into this AppDROP

class dlg.drop.**BarrierAppDROP**(***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.drop.**ContainerDROP**(***kwargs*)
A DROP that doesn't directly point to some piece of data, but instead holds references to other DROPs (its children), and from them its own internal state is deduced.

Because of its nature, ContainerDROPs cannot be written to directly, and likewise they cannot be read from directly. One instead has to pay attention to its "children" DROPs if I/O must be performed.

dataURL()
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.

exists()
Returns *True* if the data represented by this DROP exists indeed in the underlying storage mechanism

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.drop.**DirectoryContainer**(***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).

class dlg.drop.**FileDROP**(***kwargs*)
A DROP that points to data stored in a mounted filesystem.

Users can (but usually don't need to) specify both a *filepath* and a *dirname* parameter for each FileDrop. The combination of these two parameters will determine the final location of the file backed up by this drop on the underlying filesystem. When no `filepath` is provided, the drop's UID will be used as a filename. When a relative filepath is provided, it is relative to `dirname`. When an absolute `filepath` is given, it is used as-is. When a relative `dirname` is provided, it is relative to the base directory of the currently running session (i.e., a directory with the session ID as a name, placed within the currently working directory of the Node Manager hosting that session). If `dirname` is absolute, it is used as-is.

In some cases drops are created **outside** the context of a session, most notably during unit tests. In these cases the base directory is a fixed location under `/tmp`.

The following table summarizes the calculation of the final path used by the `FileDrop` class depending on its parameters:

.	filepath		
dirname	empty	relative	absolute
empty	/$B/$u	/$B/$f	/$f
relative	/$B/$d/$u	/$B/$d/$f	**ERROR**
absolute	/$d/$u	/$d/$f	**ERROR**

In the table, `$f` is the value of `filepath`, `$d` is the value of `dirname`, `$u` is the drop's UID and `$B` is the base directory for this drop's session, namelly `/the/cwd/$session_id`.

delete()
Deletes the data represented by this DROP.

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.

class dlg.drop.**InMemoryDROP**(***kwargs*)
A DROP that points data stored in memory.

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.drop.**InputFiredAppDROP**(***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 state.

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(***kwargs*)
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.

class dlg.drop.**ListAsDict**(*my_set*)
A list that adds drop UIDs to a set as they get appended to the list

append(*drop*)
L.append(object) – append object to end

class dlg.drop.**NgasDROP**(***kwargs*)
A DROP that points to data stored in an NGAS server

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.drop.**NullDROP**(***kwargs*)
A DROP that doesn't store any data.

getIO()
Returns an instance of one of the *dlg.io.DataIO* instances that handles the data contents of this DROP.

class dlg.drop.**PathBasedDrop**
Base class for data drops that handle paths (i.e., file and directory drops)

class dlg.drop.**RDBMSDrop**(***kwargs*)
A Drop that stores data in a table of a relational database

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*)
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.

class dlg.drop.**ShoreDROP**(***kwargs*)

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.drop.**dropdict**
An intermediate representation of a DROP that can be easily serialized into a transport format such as JSON or XML.

This dictionary holds all the important information needed to call any given DROP constructor. The most essential pieces of information are the DROP's OID, and its type (which determines the class to instantiate). Depending on the type more fields will be required. This class doesn't enforce these requirements though, as it only acts as an information container.

This class also offers a few utility methods to make it look more like an actual DROP class. This way, users can use the same set of methods both to create DROPs representations (i.e., instances of this class) and actual DROP instances.

Users of this class are, for example, the graph_loader module which deals with JSON -> DROP representation transformations, and the different repositories where graph templates are expected to be found by the DROP-Manager.

7.1.4 dlg.s3_drop

Drops that interact with AWS S3

class dlg.s3_drop.**S3DROP**(*oid*, *uid*, ***kwargs*)
A DROP that points to data stored in S3

bucket
Returns the bucket name :return: the bucket name

exists()
Returns *True* if the data represented by this DROP exists indeed in the underlying storage mechanism

getIO()
This type of DROP cannot be accessed directly :return:

initialize(***kwargs*)

Parameters **kwargs** – the dictionary of arguments

key
Return the S3 key :return: the S3 key

path
Returns the path to the S3 object :return: the path

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.

7.1.5 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*)
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*)
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=4096*)
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*, *target*, *bufsize=4096*)
Manually copies data from one DROP into another, in bufsize steps

dlg.droputils.**depthFirstTraverse**(*node*, *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. An 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**(*nodes*)
Returns a list of all the "leaf nodes" of the graph pointed by *nodes*. *nodes* is either a single DROP, or a list of DROPs.

dlg.droputils.**getUpstreamObjects**(*drop*)
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.**get_leaves**(*pg_spec*)
Returns a set with the OIDs of the dropspecs that are the leaves of the given physical graph specification.

dlg.droputils.**get_roots**(*pg_spec*)
Returns a set with the OIDs of the dropspecs that are the roots of the given physical graph specification.

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.**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]`.

7.1.6 dlg.utils

Module containing miscellaneous utility classes and functions.

class dlg.utils.**ZlibCompressedStream**(*content*)
An object that takes a input of uncompressed stream and returns a compressed version of its contents when .read() is read.

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.**b2s**(*b*, *enc='utf8'*)
Converts bytes into a string

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.**check_port**(*host*, *port*, *timeout=0*, *checking_open=True*, *return_socket=False*)
Checks that the port specified by host:port is either open or closed (depending on the value of checking_open) within a given timeout. When checking for an open port, this method will keep trying to connect to it either until the given timeout has expired or until the socket is found open. When checking for a closed port this method will keep trying to connect to it until the connection is unsuccessful, or until the timeout expires. Additionally, if some data is passed and the method is checking_open then data will be written to the socket if it connects successfully.

This method returns True if the port was found on the expected state within the time limit, and False otherwise.

dlg.utils.**connect_to**(*host*, *port*, *timeout=None*)
Connects to host:port within the given timeout and return the connected socket. If no connection could be established a socket.timeout error is raised

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 (assuming it's a .json file) 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.**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.**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.**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.**portIsClosed**(*host*, *port*, *timeout*)
Checks if a given host/port is closed, with a given timeout.

dlg.utils.**portIsOpen**(*host*, *port*, *timeout=0*)
Checks if a given host/port is open, with a given timeout.

dlg.utils.**prepare_sql**(*sql*, *paramstyle*, *data=()*)
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.**terminate_or_kill**(*proc*, *timeout*)
Terminates a process and waits until it has completed its execution within the given timeout. If the process is still alive after the timeout it is killed.

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.**write_to**(*host*, *port*, *data*, *timeout=None*)
Connects to `host:port` within the given timeout and write the given piece of `data` into the connected socket.

dlg.utils.**zmq_safe**(*host_or_addr*)
Converts *host_or_addr* to a format that is safe for ZMQ to use

7.1.7 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.

7.1.8 dlg.delayed

dlg.**delayed**(*x*, **args*, ***kwargs*)
Like dask.delayed, but quietly swallowing anything other than *nout*

7.2 dlg.manager

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

Contents

- *dlg.manager*
  - *dlg.manager.session*
  - *dlg.manager.drop_manager*
  - *dlg.manager.node_manager*

- *dlg.manager.composite_manager*
- *dlg.manager.rest*
- *dlg.manager.client*

7.2.1 dlg.manager.session

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

7.2.2 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.

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*.

createSession(*sessionId*)

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

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*.

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*.

7.2.3 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

class dlg.manager.node_manager.**NodeManager**(*useDLM=True*, *dlgPath=None*, *error_listener=None*, *event_listeners=[]*, *max_threads=0*, *host=None*, *rpc_port=6666*, *events_port=5555*)

class dlg.manager.node_manager.**NodeManagerBase**(*useDLM=True*, *dlgPath=None*, *error_listener=None*, *event_listeners=[]*, *max_threads=0*)
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*.

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*.

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*.

get_rpc_client(*hostname*, *port*)
Creates an RPC client connected to the node manager running in `host:port`, and its closing method, as a 2-tuple.

publish_event(*evt*)
Publishes the event `evt` for other Node Managers to receive it

shutdown()
Stops any pending background task run by this Node Manager

start()
Starts any background task required by this Node Manager

subscribe(*host*, *port*)
Subscribes this Node Manager to events published in from `host:port`

class dlg.manager.node_manager.**RpcMixIn**(*host*, *port*)

7.2.4 dlg.manager.composite_manager

7.2.5 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.

7.2.6 dlg.manager.client

class dlg.manager.client.**BaseDROPManagerClient**(*host*, *port*, *timeout*)
Base class for REST clients that talk to the DROP managers.

addGraphSpec(*sessionId*, *graphSpec*)
Appends a graph to session *sessionId*, without creating its DROPs yet, but checking that the graph looks correct

append_graph(*sessionId*, *graphSpec*)
Appends a graph to session *sessionId*, without creating its DROPs yet, but checking that the graph looks correct

createSession(*sessionId*)
Creates a session with *sessionId*

create_session(*sessionId*)
Creates a session with *sessionId*

deploySession(*sessionId*, *completed_uids=[]*)
Deploys session *sessionId*, effectively creating its DROPs and triggering the execution of the graph

deploy_session(*sessionId*, *completed_uids=[]*)
Deploys session *sessionId*, effectively creating its DROPs and triggering the execution of the graph

destroySession(*sessionId*)
Destroys session *sessionId*

destroy_session(*sessionId*)
Destroys session *sessionId*

getGraph(*sessionId*)
Returns a dictionary where the key are the DROP UIDs, and the values are the DROP specifications.

getGraphSize(*sessionId*)
Returns the size of the graph of session *sessionId*

getGraphStatus(*sessionId*)
Returns a dictionary where the keys are DROP UIDs and the values are their corresponding status.

getSessionStatus(*sessionId*)
Returns the status of session *sessionId*

graph(*sessionId*)
Returns a dictionary where the key are the DROP UIDs, and the values are the DROP specifications.

graph_size(*sessionId*)
Returns the size of the graph of session *sessionId*

graph_status(*sessionId*)
Returns a dictionary where the keys are DROP UIDs and the values are their corresponding status.

session(*sessionId*)
Returns the details of sessions *sessionId*

session_status(*sessionId*)
Returns the status of session *sessionId*

sessions()
Returns a list of all the sessions currently held by the DROP Manager

class dlg.manager.client.**CompositeManagerClient**(*host*, *port*, *timeout*)

class dlg.manager.client.**DataIslandManagerClient**(*host='localhost'*, *port=8001*, *timeout=10*)
A DataIslandManager REST client

class dlg.manager.client.**MasterManagerClient**(*host='localhost'*, *port=8002*, *timeout=10*)
A MasterManager REST client

class dlg.manager.client.**NodeManagerClient**(*host='localhost'*, *port=8000*, *timeout=10*)
A NodeManager REST client

7.3 dlg.apps

This package contains several general-purpose applications in form of DROPs that we have developed as examples and for real-life use. Most of them are based on the `BarrierAppDROP`.

Contents

- *dlg.apps*
  - *dlg.apps.bash_shell_app*
  - *dlg.apps.dynlib*
  - *dlg.apps.dockerapp*
  - *dlg.apps.socket_listener*
  - *dlg.apps.spead_receiver*
  - *dlg.apps.scp*
  - *dlg.apps.archiving*
  - *dlg.apps.crc*

7.3.1 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**(***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**(***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**(***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**(***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**(***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.

dlg.apps.bash_shell_app.**run_bash**(*cmd*, *app_uid*, *session_id*, *inputs*, *outputs*, *stdin=None*, *stdout=-1*)
Runs the given *cmd*. If any *inputs* and/or *outputs* are given (dictionaries of uid:drop elements) they are used to replace any placeholder value in *cmd* with either drop paths or dataURLs.

stdin indicates at file descriptor or file object to use as the standard input of the bash process. If not given no stdin is given to the process.

Similarly, *stdout* is a file descriptor or file object where the standard output of the process is piped to. If not given it is consumed by this method and potentially logged.

7.3.2 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**(***kwargs*)
Loads a dynamic library into the current process and runs it

run()
Run this application. It can be safely assumed that at this point all the required inputs are COMPLETED.

class dlg.apps.dynlib.**DynlibProcApp**(***kwargs*)
Loads a dynamic library in a different process and runs it

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**(***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.

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.**InvalidLibrary**

exception dlg.apps.dynlib.**finish_subprocess**

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.**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.

7.3.3 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**(***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.

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.

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*)

path
Alias for field number 0

7.3.4 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**(***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.

7.3.5 dlg.apps.spead_receiver

Module containing an (python) application that receives spead2 data

class dlg.apps.spead_receiver.**SpeadReceiverApp**(***kwargs*)
A BarrierAppDROP that listens for data using the SPEAD protocol.

This application opens a stream and adds a UDP reader on a specific host and port to listen for the data of item *itemId*. The stream is listened until it is closed. Each heap sent through the stream is checked for the item, and once found its data is written into each output of this application.

Just like the SocketListenerApp, 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.

7.3.6 dlg.apps.scp

7.3.7 dlg.apps.archiving

class dlg.apps.archiving.**ExternalStoreApp**(***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**(***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.

7.3.8 dlg.apps.crc

Module containing an example application that calculates a CRC value

class dlg.apps.crc.**CRCApp**(***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**(***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).

7.4 dlg.dropmake

Prototypical implementation of DataFlow Manager https://confluence.ska-sdp.org/display/PRODUCTTREE/C.1.2.4.4+Data+Flow+Manager The sub-modules are based on the proposed (latest) product tree as of 8 Dec 2015

Contents

- *dlg.dropmake*
  - *dlg.dropmake.web.lg_web*
  - *dlg.dropmake.pg_generator*
  - *dlg.dropmake.scheduler*
  - *dlg.dropmake.pg_manager*

7.4.1 dlg.dropmake.web.lg_web

7.4.2 dlg.dropmake.pg_generator

https://confluence.ska-sdp.org/display/PRODUCTTREE/C.1.2.4.4.2+DFM+Resource+Manager

DFM resource managr 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

Examples of logical graph node JSON representation

{ u'category': u'memory', u'data_volume': 25, u'group': -58, u'key': -59, u'loc': u'40.96484375000006 -250.53115793863992', u'text': u'Channel @

All Day'},

{ u'Arg01': u'', u'Arg02': u'', u'Arg03': u'', u'Arg04': u'', u'category': u'Component', u'execution_time': 20, u'group': -60, u'key': -56, u'loc': u'571.6718750000005 268.0000000000004', u'text': u'DD Calibration'}

exception dlg.dropmake.pg_generator.**GInvalidLink**

exception dlg.dropmake.pg_generator.**GInvalidNode**

exception dlg.dropmake.pg_generator.**GPGTException**

exception dlg.dropmake.pg_generator.**GPGTNoNeedMergeException**

exception dlg.dropmake.pg_generator.**GraphException**

class dlg.dropmake.pg_generator.**LG**(*f*, *ssid=None*)
An object representation of Logical Graph

lgn_to_pgn(*lgn*, *iid='0'*, *lpcxt=None*)
convert logical graph node to physical graph node without considering pg links

iid: instance id (string) lpcxt: Loop context

unroll_to_tpl()
Not thread-safe!

1. just create pgn anyway
2. sort out the links

class dlg.dropmake.pg_generator.**MetisPGTP**(*drop_list*, *num_partitions=1*, *min_goal=0*, *par_label='Partition'*, *ptype=0*, *ufactor=10*, *merge_parts=False*)
DROP and GOJS representations of Physical Graph Template with Partitions Based on METIS http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

get_partition_info()
partition parameter and log entry return a string

merge_partitions(*new_num_parts*, *form_island=False*, *island_type=0*, *visual=False*)
This is called during resource mapping - deploying partitioned PGT to a list of nodes

form_island: If True, the merging will form *new_num_parts* logical islands on top of existing partitions (i.e. nodes). this is also known as "reference-based merging"

If False, the merging will physically merge current partitions into *new_num_parts* new partitions (i.e. nodes) Thus, there will be no node-island 'hierarchies' created

island_type: integer, 0 - data island, 1 - compute island

to_gojs_json(*string_rep=True*, *outdict=None*, *visual=False*)
Partition the PGT into a real "PGT with Partitions", thus PGTP, using METIS built-in functions

See METIS usage: http://metis.readthedocs.io/en/latest/index.html

to_partition_input(*outf=None*)
Convert to METIS format for mapping and decomposition NOTE - Since METIS only supports Undirected Graph, we have to produce both upstream and downstream nodes to fit its input format

class dlg.dropmake.pg_generator.**MinNumPartsPGTP**(*drop_list*, *deadline*, *num_partitions=0*, *par_label='Partition'*, *max_dop=8*, *merge_parts=False*, *optimistic_factor=0.5*)

get_partition_info()
partition parameter and log entry return a string

class dlg.dropmake.pg_generator.**MySarkarPGTP**(*drop_list*, *num_partitions=0*, *par_label='Partition'*, *max_dop=8*, *merge_parts=False*)
use the MySarkarScheduler to produce the PGTP

get_partition_info()
partition parameter and log entry return a string

merge_partitions(*new_num_parts*, *form_island=False*, *island_type=0*, *visual=False*)
This is called during resource mapping - deploying partitioned PGT to a list of nodes

form_island: If True, the merging will form *new_num_parts* logical islands on top of existing partitions (i.e. nodes)

If False, the merging will physically merge current partitions into *new_num_parts* new partitions (i.e. nodes) Thus, there will be no node-island 'hierarchies' created

island_type: integer, 0 - data island, 1 - compute island

to_gojs_json(*string_rep=True*, *outdict=None*, *visual=False*)
Partition the PGT into a real "PGT with Partitions", thus PGTP

to_partition_input(*outf*)
Convert to format for mapping and decomposition

class dlg.dropmake.pg_generator.**PGT**(*drop_list*, *build_dag=True*)
A DROP representation of Physical Graph Template

dag

Return the networkx nx.DiGraph object

The weight of the same edge (u, v) also depends. If it is called after the partitioning, it could have been zeroed if both u and v is allocated to the same DropIsland

data_movement
Return the TOTAL data movement

get_opt_num_parts()
dummy for now

json
Return the JSON string representation of the PGT for visualisation

pred_exec_time(*app_drop_only=False*, *wk='weight'*, *force_answer=False*)
Predict execution time using the longest path length

to_gojs_json(*string_rep=True*, *outdict=None*, *visual=False*)
Convert PGT (without any partitions) to JSON for visualisation in GOJS

Sub-class PGTPs will override this function, and replace this with actual partitioning, and the visulisation becomes an option

to_partition_input(*outf*)
Convert to format for mapping and decomposition

to_pg_spec(*node_list*, *ret_str=True*, *num_islands=1*, *tpl_nodes_len=0*)
convert pgt to pg specification, and map that to the hardware resources

node_list: A list of nodes (list), whose length == (num_islands + num_node_mgrs) We assume that the MasterDropManager's node is NOT in the node_list

num_islands: >1 - Partitions are "conceptually" clustered into Islands 1 - Partitions MAY BE physically merged without generating islands

depending on the length of node_list

class dlg.dropmake.pg_generator.**PSOPGTP**(*drop_list*, *par_label='Partition'*, *max_dop=8*, *deadline=None*, *topk=30*, *swarm_size=40*, *merge_parts=False*)

get_partition_info()
partition parameter and log entry return a string

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.**unroll**(*lg*, *oid_prefix=None*)
Unrolls a logical graph

7.4.3 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) 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

tw - task weight dw - data weight / volume

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. Parameters ———- G : NetworkX DiGraph

Graph

weight [string (default 'weight')] Edge data key to use for weight

default_weight [integer (default 1)] The weight of edges that do not have a weight attribute

path [list] Longest path

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.**DSCScheduler**(*drop_list*)
Based on T. Yang and A. Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," in IEEE Transactions on Parallel and Distributed Systems, vol.5, no.9, pp.951-967, Sep 1994

class dlg.dropmake.scheduler.**DilworthPartition**(*gid*, *max_dop*)
Use Dilworth theorem to determine DoP see https://en.wikipedia.org/wiki/Dilworth's_theorem

The idea goes as follows: Let bpg = bipartite_graph(dag)

DoP == Poset Width == len(max_antichain) == len(min_num_chain) == (cardinality(dag) - len(max_matching(bpg)))

Note that cardinality(dag) == cardinality(bpg) / 2

See Section 3 of the paper http://opensource.uom.gr/teaching/distrubutedSite/eceutexas/dist2/ termPapers/Selma.pdf

Also http://codeforces.com/blog/entry/3781

The key is to incrementally construct the bipartite graph (bpg) from growing dag

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

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.

can_merge(*that*)

if (self._max_dop + that._max_dop <= self._ask_max_dop): return True

class dlg.dropmake.scheduler.**GraphAnnealer**(*state*, *scheduler*, *deadline=None*, *topk=None*)
Use simulated annealing for a DAG/Graph scheduling problem. There are two ways to inject constraints:

1. explicitly implement the meet_constraint function
2. add an extra penalty term in the energy function

energy()
Calculates the number of partitions

meet_constraint()
Check if the contraint is met By default, it is always met

move()
Select the neighbour, in this case Swaps two edges in the DAG if they are not the same and simply reduce by one for one of them if otherwise

class dlg.dropmake.scheduler.**KFamilyPartition**(*gid*, *max_dop*, *w_attr='num_cpus'*, *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*, *weight*)
Add a single node u to the partition

can_merge(*that*, *u*, *v*)

class dlg.dropmake.scheduler.**MCTSScheduler**(*drop_list*, *max_dop=8*, *dag=None*, *deadline=None*, *max_moves=1000*, *max_calc_time=10*)
Use Monte Carlo Tree Search to guide the Sarkar algorithm https://en.wikipedia.org/wiki/Monte_Carlo_tree_search Use basic functions in PSOScheduler by inheriting it for convinence

partition_dag()
Trigger the MCTS algorithm 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.**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.**MultiWeightPartition**(*gid*, *max_dops*, *w_attrs=['num_cpus']*, *global_dag=None*)

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

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.

can_merge(*that*)

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*)

This is called ONLY IF override_cannot_add has returned "True" 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

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)

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

schedule
Get the schedule assocaited with this partition

class dlg.dropmake.scheduler.**SAScheduler**(*drop_list*, *max_dop=8*, *dag=None*, *deadline=None*, *topk=None*, *max_iter=6000*)
Use Simulated Annealing to guide the Sarkar algorithm https://en.wikipedia.org/wiki/Simulated_annealing http://apmonitor.com/me575/index.php/Main/SimulatedAnnealing Use basic functions in PSOScheduler by inheriting it for convinence

partition_dag()
Trigger the SA algorithm 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.**Schedule**(*dag*, *max_dop*)
The scheduling solution with schedule-related properties

efficiency
resource usage percentage (integer)

schedule_matrix

Return: a self._lpl x self._max_dop matrix (X - time, Y - resource unit / parallel lane)

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=0*)

Merge M partitions into N partitions where N < M implemented using METIS for now

bal_cond: load balance condition (integer): 0 - workload, 1 - count

exception dlg.dropmake.scheduler.**SchedulerException**

class dlg.dropmake.scheduler.**WeightedDilworthPartition**(*gid*, *max_dop*, *global_dag=None*)

The extensions on DilworthPartition

Support "weights" for each Drop's DoP (e.g. *CLEAN* AppDrop uses 8 cores)

This requires a "weighted" maximal antichain. The solution is to create a weighted version of the bipartite graph without changing the original partition DAG. This allows us to use the same max matching algorithm to find the max antichain

It has an option (*global_dag*) to deal with global path reachability, which could be missing from the DAG inside the local partition. Such misses inflate DoP values, leading to more rejected partition merge requests, which in turn creates more partitions. Based on our experiment, without switching on this option, scheduling the "chiles_two_dev2" pipeline will create 45 partitions (i.e. 45 compute nodes, each has 8 cores). Turning on this option bring that number down to 24 within the same execution time.

Off - exec_time:92 - min_exec_time:67 - total_data_movement:510 - algo:Edge Zero - num_parts:45

On - exec_time:92 - min_exec_time:67 - total_data_movement:482 - algo:Edge Zero - num_parts:24

However "probing reachability" slows down partitioning by a factor of 3 in the case of the CHILES2 pipeline. Some techniques may be applicable e.g. http://www.sciencedirect.com/science/article/pii/S0196677483710175

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.

can_merge(*that*)
Need to merge split graph as well to speed up!

7.4.4 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 A:

matrix A (2d numpy array)

B: matrix B (2d numy array)

separator: whether to add an empty row separator between the two matrices (boolean)

Return: the vertically stacked matrix (2d numpy array)

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

CHAPTER 8

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

d

dlg, 27
dlg.apps, 47
dlg.apps.archiving, 53
dlg.apps.bash_shell_app, 47
dlg.apps.crc, 53
dlg.apps.dockerapp, 50
dlg.apps.dynlib, 48
dlg.apps.socket_listener, 52
dlg.apps.spead_receiver, 52
dlg.drop, 30
dlg.dropmake, 54
dlg.dropmake.pg_generator, 54
dlg.dropmake.pg_manager, 62
dlg.dropmake.scheduler, 57
dlg.dropmake.web.lg_web, 54
dlg.droputils, 38
dlg.event, 27
dlg.graph_loader, 42
dlg.io, 28
dlg.manager, 42
dlg.manager.client, 46
dlg.manager.drop_manager, 43
dlg.manager.node_manager, 44
dlg.manager.rest, 45
dlg.manager.session, 43
dlg.s3_drop, 38
dlg.utils, 40

Index

A

AbstractDROP (class in dlg.drop), 30
add() (dlg.dropmake.scheduler.DilworthPartition method), 58
add() (dlg.dropmake.scheduler.MultiWeightPartition method), 59
add() (dlg.dropmake.scheduler.Partition method), 61
add_node() (dlg.dropmake.scheduler.KFamilyPartition method), 58
add_node() (dlg.dropmake.scheduler.Partition method), 61
add_pgt() (dlg.dropmake.pg_manager.PGManager method), 62
addConsumer() (dlg.drop.AbstractDROP method), 30
addGraphSpec() (dlg.manager.client.BaseDROPManagerClient method), 46
addGraphSpec() (dlg.manager.drop_manager.DROPManager method), 43
addGraphSpec() (dlg.manager.node_manager.NodeManagerBase method), 44
addLink() (in module dlg.graph_loader), 42
addProducer() (dlg.drop.AbstractDROP method), 30
addStreamingConsumer() (dlg.drop.AbstractDROP method), 30
allDropContents() (in module dlg.droputils), 39
AppDROP (class in dlg.drop), 33
append() (dlg.drop.ListAsDict method), 36
append_graph() (dlg.manager.client.BaseDROPManagerClient method), 46

B

b2s() (in module dlg.utils), 40
BarrierAppDROP (class in dlg.drop), 34
BaseDROPManagerClient (class in dlg.manager.client), 46
BashShellApp (class in dlg.apps.bash_shell_app), 47
BashShellBase (class in dlg.apps.bash_shell_app), 47
breadFirstTraverse() (in module dlg.droputils), 39
browse_service() (in module dlg.utils), 40
bucket (dlg.s3_drop.S3DROP attribute), 38
build_dag_from_drops() (dlg.dropmake.scheduler.DAGUtil static method), 57

C

can_add() (dlg.dropmake.scheduler.DilworthPartition method), 58
can_add() (dlg.dropmake.scheduler.MultiWeightPartition method), 59
can_add() (dlg.dropmake.scheduler.Partition method), 61
can_add() (dlg.dropmake.scheduler.WeightedDilworthPartition method), 62
can_merge() (dlg.dropmake.scheduler.DilworthPartition method), 58
can_merge() (dlg.dropmake.scheduler.KFamilyPartition method), 58
can_merge() (dlg.dropmake.scheduler.MultiWeightPartition method), 59
can_merge() (dlg.dropmake.scheduler.WeightedDilworthPartition method), 62
CDlgApp (class in dlg.apps.dynlib), 48
CDlgInput (class in dlg.apps.dynlib), 48
CDlgOutput (class in dlg.apps.dynlib), 48
CDlgStreamingInput (class in dlg.apps.dynlib), 48
check_port() (in module dlg.utils), 40
checksum (dlg.drop.AbstractDROP attribute), 31
checksumType (dlg.drop.AbstractDROP attribute), 31
close() (dlg.drop.AbstractDROP method), 31
close() (dlg.io.DataIO method), 28
CompositeManagerClient (class in dlg.manager.client), 47
CompositeManagerRestServer (class in dlg.manager.rest), 45
connect_to() (in module dlg.utils), 41
constrain_func() (dlg.dropmake.scheduler.PSOScheduler method), 60
consumers (dlg.drop.AbstractDROP attribute), 31
ContainerDROP (class in dlg.drop), 34
ContainerIpWaiter (class in dlg.apps.dockerapp), 50
copyDropContents() (in module dlg.droputils), 39

CRCApp (class in dlg.apps.crc), 53
CRCStreamApp (class in dlg.apps.crc), 53
create_session() (dlg.manager.client.BaseDROPManagerClient method), 46
createDirIfMissing() (in module dlg.utils), 41
createSession() (dlg.manager.client.BaseDROPManagerClient method), 46
createSession() (dlg.manager.drop_manager.DROPManager method), 43
createSession() (dlg.manager.node_manager.NodeManagerBase method), 44

D

dag (dlg.dropmake.pg_generator.PGT attribute), 56
DAGUtil (class in dlg.dropmake.scheduler), 57
data_movement (dlg.dropmake.pg_generator.PGT attribute), 56
DataIO (class in dlg.io), 28
DataIslandManagerClient (class in dlg.manager.client), 47
dataURL() (dlg.drop.AbstractDROP method), 31
dataURL() (dlg.drop.ContainerDROP method), 34
dataWritten() (dlg.apps.bash_shell_app.StreamingInputBashAppBase method), 48
dataWritten() (dlg.apps.crc.CRCStreamApp method), 53
dataWritten() (dlg.apps.dynlib.DynlibStreamApp method), 49
dataWritten() (dlg.drop.AppDROP method), 33
decrRefCount() (dlg.drop.AbstractDROP method), 31
delayed() (in module dlg), 42
delete() (dlg.drop.AbstractDROP method), 31
delete() (dlg.drop.ContainerDROP method), 34
delete() (dlg.drop.DirectoryContainer method), 35
delete() (dlg.drop.FileDROP method), 35
delete() (dlg.io.DataIO method), 28
delete() (dlg.io.ErrorIO method), 28
delete() (dlg.io.FileIO method), 29
delete() (dlg.io.MemoryIO method), 29
delete() (dlg.io.NgasIO method), 29
delete() (dlg.io.NgasLiteIO method), 29
delete() (dlg.io.NullIO method), 29
delete() (dlg.io.ShoreIO method), 29
deliver_event() (dlg.manager.node_manager.NodeManagerBase method), 44
deploy_session() (dlg.manager.client.BaseDROPManagerClient method), 46
deploySession() (dlg.manager.client.BaseDROPManagerClient method), 46
deploySession() (dlg.manager.drop_manager.DROPManager method), 43
deploySession() (dlg.manager.node_manager.NodeManagerBase method), 44
depthFirstTraverse() (in module dlg.droputils), 39
deregister_service() (in module dlg.utils), 41
destroy_session() (dlg.manager.client.BaseDROPManagerClient method), 46
destroySession() (dlg.manager.client.BaseDROPManagerClient method), 46
destroySession() (dlg.manager.drop_manager.DROPManager method), 43
destroySession() (dlg.manager.node_manager.NodeManagerBase method), 44
DilworthPartition (class in dlg.dropmake.scheduler), 57
DirectoryContainer (class in dlg.drop), 34
dlg (module), 27
dlg.apps (module), 47
dlg.apps.archiving (module), 53
dlg.apps.bash_shell_app (module), 47
dlg.apps.crc (module), 53
dlg.apps.dockerapp (module), 50
dlg.apps.dynlib (module), 48
dlg.apps.socket_listener (module), 52
dlg.apps.spead_receiver (module), 52
dlg.drop (module), 30
dlg.dropmake (module), 54
dlg.dropmake.pg_generator (module), 54
dlg.dropmake.pg_manager (module), 62
dlg.dropmake.scheduler (module), 57
dlg.dropmake.web.lg_web (module), 54
dlg.droputils (module), 38
dlg.event (module), 27
dlg.graph_loader (module), 42
dlg.io (module), 28
dlg.manager (module), 42
dlg.manager.client (module), 46
dlg.manager.drop_manager (module), 43
dlg.manager.node_manager (module), 44
dlg.manager.rest (module), 45
dlg.manager.session (module), 43
dlg.s3_drop (module), 38
dlg.utils (module), 40
DockerApp (class in dlg.apps.dockerapp), 50
DockerPath (class in dlg.apps.dockerapp), 51
dropCompleted() (dlg.apps.bash_shell_app.StreamingInputBashAppBase method), 48
dropCompleted() (dlg.apps.crc.CRCStreamApp method), 53
dropCompleted() (dlg.apps.dynlib.DynlibStreamApp method), 49
dropCompleted() (dlg.drop.AppDROP method), 33
dropCompleted() (dlg.drop.InputFiredAppDROP method), 36
dropdict (class in dlg.drop), 37
DROPFile (class in dlg.droputils), 38
DROPManager (class in dlg.manager.drop_manager), 43
DROPWaiterCtx (class in dlg.droputils), 39
DSCScheduler (class in dlg.dropmake.scheduler), 57
DynlibApp (class in dlg.apps.dynlib), 48

DynlibProcApp (class in dlg.apps.dynlib), 49
DynlibStreamApp (class in dlg.apps.dynlib), 49

E

efficiency (dlg.dropmake.scheduler.Schedule attribute), 61
energy() (dlg.dropmake.scheduler.GraphAnnealer method), 58
ErrorIO (class in dlg.io), 28
ErrorStatusListener (class in dlg.manager.node_manager), 44
escapeQuotes() (in module dlg.utils), 41
Event (class in dlg.event), 27
EventFirer (class in dlg.event), 28
EvtConsumer (class in dlg.droputils), 39
execStatus (dlg.drop.AppDROP attribute), 33
execute() (dlg.drop.InputFiredAppDROP method), 36
executionMode (dlg.drop.AbstractDROP attribute), 31
exists() (dlg.drop.AbstractDROP method), 31
exists() (dlg.drop.ContainerDROP method), 34
exists() (dlg.drop.DirectoryContainer method), 35
exists() (dlg.drop.InputFiredAppDROP method), 36
exists() (dlg.io.DataIO method), 28
exists() (dlg.io.ErrorIO method), 29
exists() (dlg.io.FileIO method), 29
exists() (dlg.io.MemoryIO method), 29
exists() (dlg.io.NgasIO method), 29
exists() (dlg.io.NgasLiteIO method), 29
exists() (dlg.io.NullIO method), 29
exists() (dlg.io.ShoreIO method), 30
exists() (dlg.s3_drop.S3DROP method), 38
ExternalStoreApp (class in dlg.apps.archiving), 53

F

FileDROP (class in dlg.drop), 35
FileIO (class in dlg.io), 29
finish_subprocess, 49
fname_to_pipname() (in module dlg.utils), 41

G

ganttchart_matrix() (dlg.dropmake.scheduler.DAGUtil static method), 57
get_all_ipv4_addresses() (in module dlg.utils), 41
get_from_subprocess() (in module dlg.apps.dynlib), 49
get_gantt_chart() (dlg.dropmake.pg_manager.PGManager method), 62
get_leaves() (in module dlg.droputils), 40
get_local_ip_addr() (in module dlg.utils), 41
get_longest_path() (dlg.dropmake.scheduler.DAGUtil static method), 57
get_max_antichains() (dlg.dropmake.scheduler.DAGUtil static method), 57
get_max_dop() (dlg.dropmake.scheduler.DAGUtil static method), 57
get_max_width() (dlg.dropmake.scheduler.DAGUtil static method), 57
get_opt_num_parts() (dlg.dropmake.pg_generator.PGT method), 56
get_partition_info() (dlg.dropmake.pg_generator.MetisPGTP method), 55
get_partition_info() (dlg.dropmake.pg_generator.MinNumPartsPGTP method), 55
get_partition_info() (dlg.dropmake.pg_generator.MySarkarPGTP method), 55
get_partition_info() (dlg.dropmake.pg_generator.PSOPGTP method), 56
get_pgt() (dlg.dropmake.pg_manager.PGManager method), 62
get_roots() (in module dlg.droputils), 40
get_rpc_client() (dlg.manager.node_manager.NodeManagerBase method), 45
get_schedule_matrices() (dlg.dropmake.pg_manager.PGManager method), 63
getDlgDir() (in module dlg.utils), 41
getDlgLogsDir() (in module dlg.utils), 41
getDlgPidDir() (in module dlg.utils), 41
getDownstreamObjects() (in module dlg.droputils), 39
getGraph() (dlg.manager.client.BaseDROPManagerClient method), 46
getGraph() (dlg.manager.drop_manager.DROPManager method), 43
getGraph() (dlg.manager.node_manager.NodeManagerBase method), 44
getGraphSize() (dlg.manager.client.BaseDROPManagerClient method), 46
getGraphSize() (dlg.manager.drop_manager.DROPManager method), 43
getGraphSize() (dlg.manager.node_manager.NodeManagerBase method), 44
getGraphStatus() (dlg.manager.client.BaseDROPManagerClient method), 46
getGraphStatus() (dlg.manager.drop_manager.DROPManager method), 43
getGraphStatus() (dlg.manager.node_manager.NodeManagerBase method), 44
getIO() (dlg.drop.AbstractDROP method), 31
getIO() (dlg.drop.ContainerDROP method), 34
getIO() (dlg.drop.FileDROP method), 35
getIO() (dlg.drop.InMemoryDROP method), 35
getIO() (dlg.drop.NgasDROP method), 36
getIO() (dlg.drop.NullDROP method), 37
getIO() (dlg.drop.RDBMSDrop method), 37
getIO() (dlg.drop.ShoreDROP method), 37
getIO() (dlg.s3_drop.S3DROP method), 38
getLeafNodes() (in module dlg.droputils), 39
getSessionIds() (dlg.manager.drop_manager.DROPManager method), 44
getSessionIds() (dlg.manager.node_manager.NodeManagerBase

method), 44
getSessionStatus() (dlg.manager.client.BaseDROPManagerClient method), 46
getSessionStatus() (dlg.manager.drop_manager.DROPManager method), 44
getSessionStatus() (dlg.manager.node_manager.NodeManagerBase method), 45
getUpstreamObjects() (in module dlg.droputils), 39
GInvalidLink, 54
GInvalidNode, 54
GPGTException, 54
GPGTNoNeedMergeException, 54
graph() (dlg.manager.client.BaseDROPManagerClient method), 46
graph_size() (dlg.manager.client.BaseDROPManagerClient method), 46
graph_status() (dlg.manager.client.BaseDROPManagerClient method), 46
GraphAnnealer (class in dlg.dropmake.scheduler), 58
GraphException, 54

H

handleEvent() (dlg.drop.AbstractDROP method), 31
handleEvent() (dlg.drop.AppDROP method), 33
handleInterest() (dlg.apps.dockerapp.DockerApp method), 51
handleInterest() (dlg.drop.AbstractDROP method), 31
has_path() (in module dlg.droputils), 40

I

incrRefCount() (dlg.drop.AbstractDROP method), 32
initialize() (dlg.apps.archiving.NgasArchivingApp method), 53
initialize() (dlg.apps.bash_shell_app.StreamingInputBashAppBase method), 48
initialize() (dlg.apps.crc.CRCStreamApp method), 53
initialize() (dlg.apps.dockerapp.DockerApp method), 51
initialize() (dlg.apps.dynlib.DynlibProcApp method), 49
initialize() (dlg.apps.dynlib.DynlibStreamApp method), 49
initialize() (dlg.apps.socket_listener.SocketListenerApp method), 52
initialize() (dlg.apps.spead_receiver.SpeadReceiverApp method), 52
initialize() (dlg.drop.AbstractDROP method), 32
initialize() (dlg.drop.AppDROP method), 33
initialize() (dlg.drop.BarrierAppDROP method), 34
initialize() (dlg.drop.ContainerDROP method), 34
initialize() (dlg.drop.DirectoryContainer method), 35
initialize() (dlg.drop.FileDROP method), 35
initialize() (dlg.drop.InMemoryDROP method), 35
initialize() (dlg.drop.InputFiredAppDROP method), 36
initialize() (dlg.drop.NgasDROP method), 37
initialize() (dlg.drop.RDBMSDrop method), 37
initialize() (dlg.drop.ShoreDROP method), 37
initialize() (dlg.s3_drop.S3DROP method), 38
initializeSpecifics() (dlg.manager.rest.CompositeManagerRestServer method), 45
initializeSpecifics() (dlg.manager.rest.ManagerRestServer method), 45
initializeSpecifics() (dlg.manager.rest.MasterManagerRestServer method), 45
initializeSpecifics() (dlg.manager.rest.NMRestServer method), 45
InMemoryDROP (class in dlg.drop), 35
InputFiredAppDROP (class in dlg.drop), 36
inputs (dlg.drop.AppDROP attribute), 34
insert() (dlg.drop.RDBMSDrop method), 37
InvalidLibrary, 49
IOForURL() (in module dlg.io), 29
is_time_critical() (dlg.dropmake.scheduler.MinNumPartsScheduler method), 59
is_time_critical() (dlg.dropmake.scheduler.MySarkarScheduler method), 59
isabs() (in module dlg.utils), 41
isBeingRead() (dlg.drop.AbstractDROP method), 32
isCompleted() (dlg.drop.AbstractDROP method), 32

J

json (dlg.dropmake.pg_generator.PGT attribute), 56

K

key (dlg.s3_drop.S3DROP attribute), 38
KFamilyPartition (class in dlg.dropmake.scheduler), 58

L

label_schedule() (dlg.dropmake.scheduler.DAGUtil static method), 57
LG (class in dlg.dropmake.pg_generator), 54
lgn_to_pgn() (dlg.dropmake.pg_generator.LG method), 54
ListAsDict (class in dlg.drop), 36
listify() (in module dlg.droputils), 40
load_and_init() (in module dlg.apps.dynlib), 49
loadDropSpecs() (in module dlg.graph_loader), 42

M

ManagerRestServer (class in dlg.manager.rest), 45
map_partitions() (dlg.dropmake.scheduler.Scheduler method), 62
MasterManagerClient (class in dlg.manager.client), 47
MasterManagerRestServer (class in dlg.manager.rest), 45
MCTSScheduler (class in dlg.dropmake.scheduler), 58
meet_constraint() (dlg.dropmake.scheduler.GraphAnnealer method), 58
MemoryIO (class in dlg.io), 29
merge_partitions() (dlg.dropmake.pg_generator.MetisPGTP method), 55

merge_partitions() (dlg.dropmake.pg_generator.MySarkarPGTP method), 55
merge_partitions() (dlg.dropmake.scheduler.Scheduler method), 62
metis_part() (dlg.dropmake.scheduler.DAGUtil static method), 57
MetisPGTP (class in dlg.dropmake.pg_generator), 55
MinNumPartsPGTP (class in dlg.dropmake.pg_generator), 55
MinNumPartsScheduler (class in dlg.dropmake.scheduler), 59
move() (dlg.dropmake.scheduler.GraphAnnealer method), 58
MultiWeightPartition (class in dlg.dropmake.scheduler), 59
MySarkarPGTP (class in dlg.dropmake.pg_generator), 55
MySarkarScheduler (class in dlg.dropmake.scheduler), 59

N

NgasArchivingApp (class in dlg.apps.archiving), 53
NgasDROP (class in dlg.drop), 36
NgasIO (class in dlg.io), 29
NgasLiteIO (class in dlg.io), 29
NMRestServer (class in dlg.manager.rest), 45
NodeManager (class in dlg.manager.node_manager), 44
NodeManagerBase (class in dlg.manager.node_manager), 44
NodeManagerClient (class in dlg.manager.client), 47
NullDROP (class in dlg.drop), 37
NullIO (class in dlg.io), 29

O

object_tracking() (in module dlg.utils), 41
objective_func() (dlg.dropmake.scheduler.PSOScheduler method), 60
oid (dlg.drop.AbstractDROP attribute), 32
open() (dlg.drop.AbstractDROP method), 32
open() (dlg.io.DataIO method), 28
outputs (dlg.drop.AppDROP attribute), 34
override_cannot_add() (dlg.dropmake.scheduler.MinNumPartsScheduler method), 59
override_cannot_add() (dlg.dropmake.scheduler.MySarkarScheduler method), 60

P

parent (dlg.drop.AbstractDROP attribute), 32
Partition (class in dlg.dropmake.scheduler), 60
partition() (in module dlg.dropmake.pg_generator), 56
partition_dag() (dlg.dropmake.scheduler.MCTSScheduler method), 58
partition_dag() (dlg.dropmake.scheduler.MySarkarScheduler method), 60
partition_dag() (dlg.dropmake.scheduler.PSOScheduler method), 60
partition_dag() (dlg.dropmake.scheduler.SAScheduler method), 61
path (dlg.apps.dockerapp.DockerPath attribute), 51
path (dlg.s3_drop.S3DROP attribute), 38
PathBasedDrop (class in dlg.drop), 37
PGManager (class in dlg.dropmake.pg_manager), 62
PGT (class in dlg.dropmake.pg_generator), 56
PGUtil (class in dlg.dropmake.pg_manager), 63
phase (dlg.drop.AbstractDROP attribute), 32
portIsClosed() (in module dlg.utils), 41
portIsOpen() (in module dlg.utils), 41
precious (dlg.drop.AbstractDROP attribute), 32
pred_exec_time() (dlg.dropmake.pg_generator.PGT method), 56
prepare_c_inputs() (in module dlg.apps.dynlib), 49
prepare_c_outputs() (in module dlg.apps.dynlib), 49
prepare_input_channel() (in module dlg.apps.bash_shell_app), 48
prepare_output_channel() (in module dlg.apps.bash_shell_app), 48
prepare_sql() (in module dlg.utils), 41
probe_max_dop() (dlg.dropmake.scheduler.Partition method), 61
producerFinished() (dlg.drop.AbstractDROP method), 32
producers (dlg.drop.AbstractDROP attribute), 32
prune_antichains() (dlg.dropmake.scheduler.DAGUtil static method), 57
PSOPGTP (class in dlg.dropmake.pg_generator), 56
PSOScheduler (class in dlg.dropmake.scheduler), 60
publish_event() (dlg.manager.node_manager.NodeManagerBase method), 45

R

RDBMSDrop (class in dlg.drop), 37
read() (dlg.drop.AbstractDROP method), 32
read() (dlg.io.DataIO method), 28
register_service() (in module dlg.utils), 42
remove() (dlg.dropmake.scheduler.Partition method), 61
replace_dataurl_placeholders() (in module dlg.droputils), 40
replace_path_placeholders() (in module dlg.droputils), 40
RpcMixIn (class in dlg.manager.node_manager), 45
run() (dlg.apps.archiving.ExternalStoreApp method), 53
run() (dlg.apps.bash_shell_app.BashShellApp method), 47
run() (dlg.apps.bash_shell_app.StreamingOutputBashApp method), 48
run() (dlg.apps.crc.CRCApp method), 53
run() (dlg.apps.dockerapp.DockerApp method), 51
run() (dlg.apps.dynlib.DynlibApp method), 49
run() (dlg.apps.dynlib.DynlibProcApp method), 49

run() (dlg.apps.socket_listener.SocketListenerApp method), 52
run() (dlg.apps.spead_receiver.SpeadReceiverApp method), 52
run() (dlg.drop.InputFiredAppDROP method), 36
run() (in module dlg.apps.dynlib), 49
run_bash() (in module dlg.apps.bash_shell_app), 48

S

S3DROP (class in dlg.s3_drop), 38
SAScheduler (class in dlg.dropmake.scheduler), 61
Schedule (class in dlg.dropmake.scheduler), 61
schedule (dlg.dropmake.scheduler.Partition attribute), 61
schedule_matrix (dlg.dropmake.scheduler.Schedule attribute), 61
Scheduler (class in dlg.dropmake.scheduler), 61
SchedulerException, 62
select() (dlg.drop.RDBMSDrop method), 37
session() (dlg.manager.client.BaseDROPManagerClient method), 46
session_status() (dlg.manager.client.BaseDROPManagerClient method), 46
sessions() (dlg.manager.client.BaseDROPManagerClient method), 46
setCompleted() (dlg.drop.AbstractDROP method), 32
setError() (dlg.drop.AbstractDROP method), 33
ShoreDROP (class in dlg.drop), 37
ShoreIO (class in dlg.io), 29
shutdown() (dlg.manager.node_manager.NodeManagerBase method), 45
size (dlg.drop.AbstractDROP attribute), 33
size() (dlg.s3_drop.S3DROP method), 38
SocketListenerApp (class in dlg.apps.socket_listener), 52
SpeadReceiverApp (class in dlg.apps.spead_receiver), 52
start() (dlg.manager.node_manager.NodeManagerBase method), 45
status (dlg.drop.AbstractDROP attribute), 33
store() (dlg.apps.archiving.ExternalStoreApp method), 53
store() (dlg.apps.archiving.NgasArchivingApp method), 53
streamingConsumers (dlg.drop.AbstractDROP attribute), 33
StreamingInputBashApp (class in dlg.apps.bash_shell_app), 47
StreamingInputBashAppBase (class in dlg.apps.bash_shell_app), 48
StreamingInputOutputBashApp (class in dlg.apps.bash_shell_app), 48
streamingInputs (dlg.drop.AppDROP attribute), 34
StreamingOutputBashApp (class in dlg.apps.bash_shell_app), 48
subscribe() (dlg.event.EventFirer method), 28
subscribe() (dlg.manager.node_manager.NodeManagerBase method), 45

T

terminate_or_kill() (in module dlg.utils), 42
to_externally_contactable_host() (in module dlg.utils), 42
to_gojs_json() (dlg.dropmake.pg_generator.MetisPGTP method), 55
to_gojs_json() (dlg.dropmake.pg_generator.MySarkarPGTP method), 56
to_gojs_json() (dlg.dropmake.pg_generator.PGT method), 56
to_partition_input() (dlg.dropmake.pg_generator.MetisPGTP method), 55
to_partition_input() (dlg.dropmake.pg_generator.MySarkarPGTP method), 56
to_partition_input() (dlg.dropmake.pg_generator.PGT method), 56
to_pg_spec() (dlg.dropmake.pg_generator.PGT method), 56

U

uid (dlg.drop.AbstractDROP attribute), 33
unroll() (in module dlg.dropmake.pg_generator), 56
unroll_to_tpl() (dlg.dropmake.pg_generator.LG method), 55
unsubscribe() (dlg.event.EventFirer method), 28

V

vstack_mat() (dlg.dropmake.pg_manager.PGUtil static method), 63

W

WeightedDilworthPartition (class in dlg.dropmake.scheduler), 62
workload (dlg.dropmake.scheduler.Schedule attribute), 61
write() (dlg.drop.AbstractDROP method), 33
write() (dlg.io.DataIO method), 28
write_to() (in module dlg.utils), 42

Z

ZlibCompressedStream (class in dlg.utils), 40
ZlibUncompressedStream (class in dlg.utils), 40
zmq_safe() (in module dlg.utils), 42