AIDA Developments Guidelines

20 February 2013 - Lausanne / Boston

1. General structure and notes

AIDA is organized with a well defined modular structure, reflected in the code modules and in the API calls. The main structure is reported in the following picture, showing the main blocks and the interactions among them. This is a schematic picture, with no porpouse to be exhaustive or complete, but it's a general map to understand how the code has been organized and how to extend it with plugins and other modules.

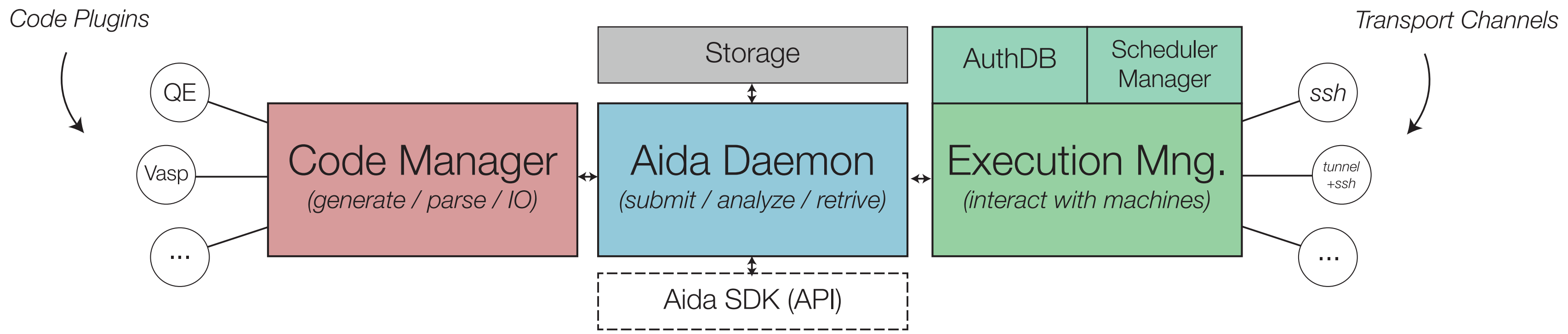

Figure 1: AIDA Modular Structure

Versions

Two different version of AIDA are present, a server versions (AIDA Server) and a personal stand-alone version (AIDA Personal). The code, API and modules will be identical in the logical structure, and configuration file will be portable among the versions, but the AIDA Daemon implementation details will change. In the following documentation only the Server version is discussed, leaving the Personal version details to another document.

License

AIDA is a Python project, requiring Python versions > 2.7. The code will be released under the **original Berkeley Software Distribution** (BSD), and all the modules and plugins distributed with AIDA and necessary for it's correct functioning have to be chosen in compliant with the respective licenses.

2. Modules definitions and methods

In the following chapters all the main modules of Fig.1 are described and the main methods and calls are listed. To better understand all the documentation it's important to intrude definitions of some terms used in this document, reflecting some logical building blocks of the entire project.

Calculation: a calculation is a numerical simulation uniquely defined by it's input parameters, software and version of the software needed, computational details of the execution and location of the execution (which machine has been used). With this informations a calculation will produce (in principle) the same identical output.

Input parameters: ...

Execution Directives: ...

With these definitions in mind we can introduce modules and methods building AIDA. As a final note, in AIDA there will be one exception type for each module, with different messages identifying the motivation of the failure. In rare cases, for critical functions, specific Exceptions will be encoded to better deal with them.

2.1 AIDA Daemon

The AIDA Daemon is the main heart of the platform, coordinating all the modules and offering all the API functions for the user to interact with.

- The API endpoints will be accessible from the server's shell and through HTTP REST protocol, both requiring an authentication scheme that will allow all the other modules to interact with external machine and databases with correct privileges (more on this later). From the API a user can send to the AIDA platform a calculation request, where all the necessary parameters have to be defined (either new parameters or already present in the platform DB), control the executions taking place and interact with the storage to retrive, modify or add new entries.

- Every time a calculation have to be executed, the Daemon will interact with the Code Manager to generate all the input files and directive necessary to execute the Job. The Daemon will pass to the Code Manager only the informations given by the user, after storing them in the Database, and will obtain a sandbox necessary to call the Execution Manager. The sandbox will be partially stored by the Daemon in the DB and the procedure can continue.

- Once the Input sandbox has been generated, the Daemon can call the Execution Manager sending both the sandbox and the Execution Directives to start an executions. The Daemon will also interact with the Execution Manager to periodically check the results of the calculations, and to control the executions upon a user request through the APIs. The Daemon will query the Execution Manager only through the UUID of the calculation, being the index of the DB.

- The Daemon is the only module interacting directly with the Storage, and all the calls necessary to save and retrieve data will be handled for the user, and the other modules, by the Daemon. With this first version the Daemon will not have any analytics feature,

2.1 Execution Manager

The Execution Manager is a middle layer between the Daemon and the execution machines. One of it's main objectives is translating the more abstract calls, such as "run the calculation" to practical procedures depending on the machine selected and the type of calculations. For each new calculation it will receive in input the Code Manager sandbox output and computations directives, and it will instantiate the run. To do this the Execution Manager will first run the Scheduler Manager asking it to produce correct launching scripts for the Queue manager present in the target machine, and that it will ask the Transport Channel to upload the necessary files and start the calculations. The Execution Manager will also handle all the queries coming from the Daemon, dispatching specific queries commands generated by the Scheduler Manager to the correct machine, retrieving results and giving back data to the Daemon.

Notes: ExecManager must access to resources only through UUID + resource name + authentication. If a user wants to get infos through a jobid, note that it exists a 1<->1 correspondence between jobid and UUID. Once you build this linking functions it's done. And, if things are well done, the user should only query through UUIDs. The authorization issue. Shouldn't be a huge problem when trying to modify something on the cluster. Since you are running locally, you will try to ssh to a cluster with any username if you want, but since the ssh key is not configured, it will fail. However: the database is exposed to anyone sharing the use if Aida the keys can stay in a folder keys / or tokens or a different database or just files that will make the correspondence between aida users and remote users and link for the folder keys.

2.1.1 Execution Manager / Authorization DB

TODO--------------

2.1.2 Execution Manager / Scheduler Manager

The scheduler manager takes care of generating specific directive, commands and launch files for some specific queue manager implementations. The initial queues supported will be PBSPro, Torque, Slurm, LoadLeveler and SGE and the modules will have to correctly handle pre_exec / post_exec_script, scheduler params (walltime, other flags / mail), mpiparams, executables params, stdin/out/err and qsub (or other commands) technicalities. In general, we should send part of the Object Resources to scheduler, so that things like numcores, numnodes are known to the scheduler. So that the initializations may look like:

```
s = Scheduler(resources)
   type(s) -> PBSsch
```

Notes: the job status cannot be dependent by the queue manager. Therefore, we have to define a list of possible status, taking inspiration from the definitions of DMRAA / SAGA admitted job statuses: DRMAA API 1.0. For Example a class JobState with specific fixed values UNDETERMINED='undetermined' - QUEUED_ACTIVE='queued_active' - SYSTEM_ON_HOLD='system_on_hold' - USER_ON_HOLD='user_on_hold' - USER_SYSTEM_ON_HOLD='user_system_on_hold' - RUNNING='running' SYSTEM_SUSPENDED='system_suspended' - USER_SUSPENDED='user_suspended' - USER_SYSTEM_SUSPENDED='user_system_suspended' DONE='done' - FAILED='failed'

Methods

- `def get_submit_script()`: return object BaseClass + part scheduler dependent
- `get_queue_command(jobid=Nonde)` : # calls qstat on remote machine
- `parse_queue_out()` : parse what’s returned by qstat, returns object: QueueInfo

- `class QueueInfo`: properties: jobid, user, queue, wall, status, numnodes, numcpu
- `class BaseClass`: (more code / cluster dependent, while part are more common to a kind of scheduler (like PBS))
  returns a text:
  """#!/bin/bash -l   # note that this string could be configurable.
  … : prepend resource requests
  … : code dependent parts (module load …)
  … : Environment variables
  … :
  [mpirun + str] [exec] [args] { [< stdin] } { [> stdout] } { [2>&1] }
  echo $? > .aida / statuscode   # line to return the exit code of the program if present, or if the code doesn't return any code, a code given by the plugin
  """
- `def get_script_main_content()` : the number of cores / numnodes must come from the resources

2.1.3 Execution Manager / Transport Channel

The transport channels will handle all the low level interaction between the Execution Manager and specific machine (HPC or even single machine). The main objective is to translate abstract calls the Execution Manager need into practical system-specific OS directive. The Transport Channel will handle both the communication procedures (opening of the channel, connection handling, buffering, ecc...) and the command executions on the machine, ensuring the complete execution of Execution Manager directives. The Transport Channel must support different transport channels, as ssh, local (etc.). The selection of the transport channel must depend on the source and destination protocols (see wrapper copyfile). For example:

```
if source.protocol == destination.protocol
      TransportChannel.copy
else
      source.TransportChannel
```

Notes: Paths should be passed only by absolute path with URL like format.

Methods

- `def get()` : Retrieve file # src = remote_folder_path, dst = local_folder_path
- `def put()` : send file
- `def copy()` : Possibility of copying without going through AIDA. With Wildcard in source (=> dest is directory) Source and destination either file or directories Future (options for symbolic links, permissions) (see python functions)
- `def exec(location)` : # (see os.path)
- `def is_dir(), def is_file(), def list_dir(), def mkdir(), def remove()`
- `def set_mod()`: To control permissions (chmod)
- `def get_mod()` : To control permissions (chmod):
- `def __enter__` : # to open a transport channel open with the style of python open(), and therefore also def `__exit__`