An Alternative Approach to Configure Permanent Tasks in Farm Nodes

Markus Frank, Clara Gaspar, Beat Jost

Introduction

The current handling of the permanent processes on the data processing nodes in LHCb experiment (hlt[a-m][0-9][0-9], mona08, mona09, mona10, storerecv, etc) is based on sending commands to the *pcSrv* process executing on each of the corresponding controls PCs. This system, which is in place since roughly 10 years, has worked reliably but unfortunately has a few flaws, which are difficult to overcome:

- The entire system depends on the presence of the controls PC.
- All configuration is solely programmed to the *pcSrv* at boot time. If the *pcSrv* is restarted the information is lost. Individual *pcSrv* configurations would have to be provided for each controls PC and hence would be cumbersome. The *pcSrv* process adds all the processes to the task manager, regularly checks their existence and on death restarts the processes.
- Externally it was not easily possible to check if all tasks on a given type of nodes were present or not. Such monitoring information typically was added externally in the form of XML files and then combined to the information shown e.g. in the *farmStatus* monitor.

Though, this system also has distinct advantages:

- The definition of a class of nodes was quite simple: a node only had to match a regular expression.
- Since the whole procedure was implemented as a script, derivations of node types were simple. Sets of tasks to be started e.g. on a sub-farm were triggered by one or several function calls.
- Edits were relatively simple.

To overcome the deficiencies and to have one single source of information which processes should execute on a given node, the idea is to introduce a database driven approach as it was present in the past e.g. in the ALEPH or DELPHI experiment[1].
To improve the flexibility and the reliability of the current system we want to introduce a similar, data driven system taking advantage of the existing FMC tools. However, in addition the proposed change would allow to accomodate further functionality.

Desired Extended Functionality

Evolving from the existing use case, a more elaborated functionality should be implemented containing (as examples) the following actions:

- At BOOT: Determine which processes should be started on a given node
  Input: host name, Output: task list with startup arguments
- At any time: kill/restart all tasks of a given type on the farm
  - On a specific node
  - On a node pattern such as hlta05*, hlt[a-f][0-3][0-9]...
- At any time: kill/restart all tasks on a node
  - On a specific node
  - On a node pattern such as hlta05*, hlt[a-f][0-3][0-9]...

Clearly this list is not exhaustive, but given the basic design, as it is briefly described in the following, it should be straight forward to implement further functionalities. An example could even include a reimplementation of the existing *pcSrv* process, which would be far more flexible if linked directly to the data rather than being driven by a quite static configuration file.

1 Please Note: When talking about processes executing on a given node, intrinsic operating system processes are not the primary target of this approach. These are started and managed by the operating system, not by external utilities. The intention is to only act on processes provided by LHCb.

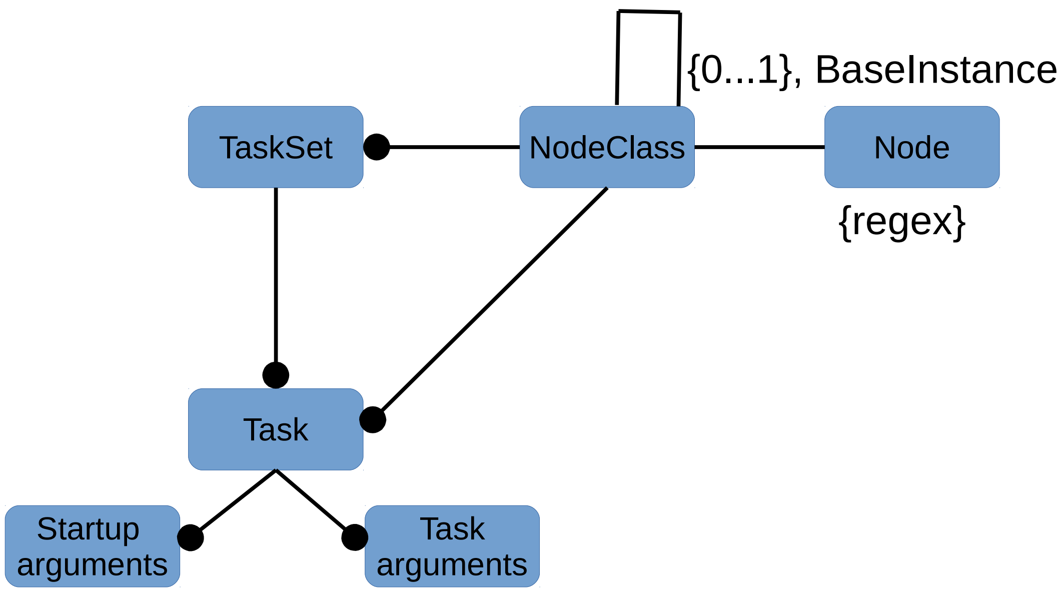

Figure 1: The basic schema to determine the processes running on a given node.

Design

As shown in Figure 1 we envisage a data driven model where every node participating in the dataflow is associated to a *NodeClass*, where a *NodeClass* may inherit for extension purposes from exactly one other *NodeClass*. To each *NodeClass* one or several *TaskSets* are associated each describing one or several processes. Which *NodeClass* should be applied for a given node depends on the match of the node name with a set of regular expressions.
All relevant items will have a description field. The process arguments will contain additional information such as when to be started, how often to restart in the event of failure, to generate an alarm on failure etc. The brief database schema in Figure 1 shall be extended and completed with evolving functionality.
Although the functionality of the design is targeted to act on nodes participating in the dataflow, it is not limited to dataflow nodes. Effectively any process to be executed on a Linux based node could be managed.

Basic API

The desired basic API should include all actions to

- Create/modify/delete/extract *Node* instances
- Create/modify/delete/extract *NodeClass* instances
- Create/modify/delete/extract *TaskSet* instances and
- Create/modify/delete/extract *Task* instances

This functionality should be accessible both from command line tools and in the more distant future from a web based tool. This API can support all higher level functionality described above.

Implementation

An initial implementation of the required database schema as well as some tools should be performed by a summer student who should:

- Refine and implement the database schema, which was briefly sketched above.
- Define the full functionality of the API
- Implement the command line API

- Implement a prototype of the boot sequence
- Start implementing a WEB editor service for the database

It is understood that the above described functionality cannot be fully implemented by a summer student. However, basic studies concerning the feasibility and practicability of such an approach can be verified. Missing and extended functionality may be added in the future.

Implementation Details

The desired functionality is not complicated and interaction with such a database schema is limited. Hence, a python based implementation is proposed. As an development database SQLite is suggested where the python database interface easily can adapt to any other relational implementation such as MySQL or Oracle. The database should be accessed using an RPC mechanism which is XML-RPC and/or DIM-RPC based to not bind any application to the database internals and the schema.