Supporting information for GLAS : An open-source easily expandable Git-based Scheduling Architecture for integral Lab Automation

Jean-Charles Cousty,[a,†] * Tanguy Cavagna,[b,†] Alec Schmidt,[b,†] Edy Mariano,[a] Keyan Villat,[a] Florian de Nanteuil,[c] Pascal Miéville [a]

[a] Swiss Cat+ West Hub, Ecole Polytechnique Fédérale de Lausanne EPFL, 1015 Lausanne, Switzerland
[b] Département Informatique et systèmes de communication, Haute école du paysage, d'ingénierie et d'architecture HEPIA,, 1202 Geneva, Switzerland
[c] DSM-Firmenich, 1217 Meyrin Switzerland

[†] Authors contributed equally

* Communication address: jean-charles.cousty@epfl.ch

November 4, 2024

S1 : Node and Workflow examples for the OmniFire

Node configuration example (PF 3400)	Workflow configuration example
{ "id": "benchbot", "name": "BenchBot", "type": "BENCHBOT", "args": { "profile": 2, "ip": "192.168.1.20", "port": "10100", "timeout": 30, "config-file": "/flash/data/omnifire.gpo" } }	{ "name": "plate analysis", "steps": ["omni-in", "benchbot", "dispatchor", "benchbot", "omni-out"] }

S2 : Multithread workflow example for the OmniFire :

```
def plate_to_stocking(self, src: Self, dst: Self, task_id: str, workflow:
Workflow, save: bool) ->

None:
       """Takes a plate from the Omni-in and stores it in the Stocking bay. Once
the plate
       is stored, pauses the thread until a notify is received. When a notify is
received,
       sleeps a second to make sure any potential incoming plate goes through
first,
       thus avoiding two plates going in at the same time. Then, grabs the plate
and
       put it on the Dispatcher before returning.

       When a plate is stored, the node's mutex is released, allowing the FP
3400
       to be used by other tasks.

       Args:
           src (Self): Previous node of the task's current node. Only to set it
```

```
as available if
                        necessary.
            dst (Self): Following node of the task's current node. This node's
state is the condition
                        variable.
            task_id (str): Id of the task running. For visual and logging
purposes.
            workflow (Workflow): Workflow object used by the task running. For
visual and logging
                                 purposes.
            save (bool): Boolean activating or disabling logging. Should inherit
_execute()'s 'save'
                         argument.

        Raises:
            BenchBotException: Something went wrong ; the node state is set to
error before raising
        """

        with self.cv_dispatchor:

            start = time.time()
            stocking_pod_id = self.store_plate(task_id, workflow.name)

            if src is not None:
                src.set_available()

            if save:
                LoggingManager.insert_data_sample(task_id, workflow.id, self.id,
start, time.time())

            start = time.time()
            self.mutex_unlock()

            """
            Must wait for the dispatcher to be available.
            When woken up, sleep one second to give time for any plate in
            the omni-in to go through first to avoid both going through.
            If a plate on the omni-in went through, go back to the passive
waiting.
            """
            while True:
                while not dst.is_available():
                    self.cv_dispatchor.wait()
                time.sleep(1) # this sleep is to allow other tasks needing the
dispatcher to go first

                if dst.is_available():
                    break

            self.mutex_lock()

            if save:
                LoggingManager.insert_data_sample(task_id, workflow.id, "Stocking
Bay", start,

time.time())
```

```
            start = time.time()
            self.pop_plate(stocking_pod_id)

            if save:
                LoggingManager.insert_data_sample(task_id, workflow.id, self.id,
start, time.time())

        dst.set_in_use()
```

S3 : A Custom base node implementation that does the same as its parent class BaseNode, except it does not update the state.

```
import time
from typing import Self, Optional

from typing_extensions import override

from glas.database.connector import DatabaseConnector
from glas.database.node_call_record import DBNodeCallRecord
from glas.logger import LoggingManager
from glas.nodes.base import BaseNode
from glas.workflow.core import Workflow


class CustomBaseNode(BaseNode):
   def __init__(self, _id: str, name: str, emulate: bool) -> None:
       self.emulate = emulate
       super().__init__(_id, name)

   @override
   def execute(self, db: DatabaseConnector, task_id: str, workflow: Workflow,
src: Self, dst: Self,
               args: dict[str, any] = None, save: bool = True) -> tuple[int, str
| None]:
       """
       Override the BaseNode execute to remove the node state update. State
updates occur:\n
           By the SCARA when it puts a plate on (sets as IN_USE)\n
           By the SCARA when it picks a plate (sets as AVAILABLE)\n
           By the Orchestrator, when the Robot Scheduler picks a plate up (sets
as AVAILABLE)\n
           By the Orchestrator, when the Robot Scheduler deposits a plate (sets
as IN_USE)
       """
       start = time.time()

       with self.mu:
           # save the access waiting time
           if save:
               LoggingManager.insert_data_sample(task_id, workflow.id, "w. acc.",
start,time.time())

           task_logger = LoggingManager.get_logger(f"task-{task_id}")
           task_logger.debug(f"Executing {self}...")
```

```
            start = time.time()

            self._pre_execution(task_id, workflow.name, src, dst, args)
            status, message, endpoint = self._execute(src, dst, workflow, task_id,
save, args)
            self._post_execution(status, message, task_id, workflow.name, src,
dst, args)

            if status != 0:
                DBNodeCallRecord.insert(db, self.id, endpoint, message,
time.time() - start, "error")
                return status, message

            DBNodeCallRecord.insert(db, self.id, endpoint, message, time.time() -
start, "success")

            if save:
                LoggingManager.insert_data_sample(task_id, workflow.id, self.id,
start, time.time())

            return 0, None

   def _execute(self, src: Self, dst: Self, workflow: Workflow, task_id: str,
save: bool,
                 args: Optional[dict[str, any]] = None) -> tuple[int,
Optional[str], Optional[str]]:
       """
       Node specific implementation of its own execution logic

       :param src: Source node
       :param dst: Destination node
       :param task_id: Caller task's id
       :param args: Execution arguments
       :return:
       """
       raise NotImplementedError("_execute")
```