Executor Framework for DIRAC

A. Casajus Ramo¹, R. Graciani Diaz¹
¹ ICC, University of Barcelona
E-mail: adria@ecm.ub.es

Abstract.
DIRAC framework for distributed computing has been designed as a group of collaborating components, agents and servers, with persistent database back-end. Components communicate with each other using DISET, an in-house protocol that provides Remote Procedure Call (RPC) and file transfer capabilities. This approach has provided DIRAC with a modular and stable design by enforcing stable interfaces across releases. But it made complicated to scale further with commodity hardware.

To further scale DIRAC, components needed to send more queries between them. Using RPC to do so requires a lot of processing power just to handle the secure handshake required to establish the connection. DISET now provides a way to keep stable connections and send and receive queries between components. Only one handshake is required to send and receive any number of queries. Using this new communication mechanism DIRAC now provides a new type of component called Executor. Executors process any task (such as resolving the input data of a job) sent to them by a task dispatcher. This task dispatcher takes care of persisting the state of the tasks to the storage backend and distributing them among all the Executors based on the requirements of each task.

In case of a high load, several Executors can be started to process the extra load and stop them once the tasks have been processed. This new approach of handling tasks in DIRAC makes Executors easy to replace and replicate, thus enabling DIRAC to further scale beyond the current approach based on polling agents.

1. Introduction
DIRAC [6] framework for distributed computing provides the means to build flexible and scalable computing systems by means of aggregating a number of DIRAC systems. Each DIRAC system contains a set of components that provide a well defined functionality to the complete system. Each DIRAC system is built using three component types: services, agents and databases [3]. Services are passive components that expect queries from another components. On the other hand, Agents are components that actively monitor the state of DIRAC and react accordingly. Databases provide a shared memory between services and agents. DISET [2] is DIRAC’s secure transport. It provides all the communication, authentication, authorization mechanisms DIRAC uses to communicate between components.

Agents loop continuously over a resource. They request the resource state and react to its state. If there are no actions to take, they will sleep for a configurable time and try again. While this mechanism is suited for some scenarios, it has some problems when polling a database. Most of DIRAC Agents poll a database to discover tasks to be executed. While this approach is rather effective when Agents are remote to the DIRAC system with an unpredictable execution time (for instance the Job Agent running on a worker node trying to match and execute a user job), it has
some drawbacks when the agents are executed centrally. In that scenario reducing the response
time is a priority. In order to improve the response time for central DIRAC components while
maintaining a low load on the system, a complementary component to the Agent framework has
been developed. This new component is called the Executor framework. Executors will register
to a central component and wait for new tasks to be sent to them instead oflooping and polling
continuously. This central component is the only one with access to the database, it retrieves
and caches the required data and sends it to the Executors to be processed. Using this new
strategy, central processing can be event driven and thus much more responsive than the current
solution without adding the extra load to the system.

DIRAC’s Workload Management System depends heavily on Agents to check jobs before they
can be executed. Each job has to be prepared based on their requirements. Each Agent checks
for a certain set of requirements or properties. For instance there is an Agent checking that the
data required by a job exists, another checks that the site requirements allows the job to run...
By isolating each group of checks in individual Agents, if a job doesn’t have the property or
requirement that triggers the Agent, it can be skipped. But each Agent has to continuously query
the database looking for jobs in a certain state. On top of that, DIRAC Workload Management
was not able to run more than one instance of each Agent because of the way the database access
implementation. Jobs can be accesses and modified by any DIRAC component at any time, if
two Agents would be started they could overwrite each other. This two problems have limited
the amount of jobs DIRAC could run in a day.

The Executor framework was designed to overcome this problems. Section 2 will introduce
the Executor framework. Section 3 will describe the new developments added to DISET that the
Executor framework uses. Section 4 will present the first usage of the Executor framework in
DIRAC. Finally section 5 will summarize the paper and outline future work.

2. Executor framework

Up until now DIRAC systems have been built on top of Services, Agents and Databases. Agents
are the main active components. They loop over a resource or a database checking for state
changes that trigger an action. If there is no action to execute, Agents will sleep for a configurable
amount of time before trying again. While this mechanism is effective when the Agent is running
remote to DIRAC with an unpredictable execution time, it has some drawbacks. The response
time will depend on the sleep time between each Agent cycle. In order to reduce the response
time the sleep time has also to be reduced, thus increasing the load on the resource or database
that it is being monitored. In order to improve the response time without adding extra load to
the system, the Executor framework has been developed.

2.1. Requirements

Before designing the Executor framework several requirements were set to allow DIRAC to
overcome the current limitations:

- Single point write access to the database. Writes can be streamlined and reordered. Only
  one components needs access to the write-enabled database. The rest of the components
  can read from read-only database slaves.
- Worker processes have to be stateless. No data should be lost if a worker process dies.
- It has to support adding and removing any worker process at any time. It has to be possible
to adapt the number of worker processes based on the task accumulation. If there is not
enough task throughput more workers will have to be added. And if some of them stay idle
too much time it has to be possible to remove them.
2.2. Design

The Executor framework is designed around two components. The Executor Mind knows how to retrieve, store and dispatch tasks. And Executors are the working processes that know what to do depending on the task type. Each Executor is an independent process that connects to the Mind and waits for tasks to be sent to them by the Mind. The mechanism used to connect the Executors to the Mind is described in section 3. A diagram of both components can been seen in figure 1.

The Mind is a DIRAC service. It is the only component of the Executor framework that needs write-access to the database. It loads tasks from the database and writes the results back. The Mind can periodically query a database to find new tasks, but it can also receive new tasks from other components. Executors don’t keep or store the result of any task. If an Executor dies without having finished a task, the Mind will simply send the task to another Executor.

When the Mind receives a task that has been properly processed by an Executor, the result will have to be stored in the database. But before storing it in the database the Mind needs to check that the task has not been modified by anyone else while the executor was processing it. To do so, the Mind has to store a task state in memory and check that this task state has not been modified before committing the result back to the database. The task state will be different for each type of task and has to be defined in each case.

When an Executor process starts it will connect to the Mind and send a list of task types it can process. The Mind acts as task scheduler and dispatcher. When the Mind has a task to be processed it will look for an idle Executor that can process that task type. If there is no idle Executor or no Executor can process that task type, the Mind will internally queue the task in memory. As soon a an Executor connects or becomes idle, the Mind will pop a task from one of the queues that the Executor can process and send the task to it. If the Executor manages to process the task, the Mind will store back the result of the task and then it will try to fill the Executor again with a new task. If the Executor disconnects while processing a task, the Mind will assume that the Executor has crashed and will reschedule the task to prevent any data loss.

Tasks may need to go through several different steps before being completely processed. This can easily be accomplished by having one task type for each step the task has to go through.
Each Executor can then publish what task types it knows how to process. For each step the task has to go through, the Mind will send the task to an Executor that can process that type of task, receive and store the result, change the task to the next type and then send the task to the next Executor. The Mind will repeat this mechanism until the task has gone through all the types.

This architecture allows to add and remove Executors at any time. If the removed Executor was being processing a task, the Mind will send the task to another Executor. If the task throughput is not enough Executors can be started and the Mind will send them tasks to process. Although Executors can be added and removed at any time, the Mind is still a single point of failure. If the Mind stops working the whole system will stop working.

3. DISET stable connections

DISET [2] is the communication, authorization and authentication framework of top of which DIRAC services are built. Traditionally DISET offered RPC [4] and file transfer capabilities. Those communication mechanisms are not well suited for the Executor framework. RPC doesn’t allow the server to send data to the clients asynchronously, and each RPC query requires establishing a new connection and going through another SSL [5] handshake. On average the SSL process is the most resource consuming part of the request.

The Executor framework relies on a new DISET capability. Support for stable connections and asynchronous requests has been added. Any component can open a connection and reuse it to send and receive requests though it. Services can send information to clients without having to wait for the clients to ask for them as shown in figure 2. Requests are encoded with DEncode, a serialization library very similar to BEncode [1] but with added serialization capabilities. BEncode does not allow serialization of dictionaries with integer as keys, None and float values and timestamps. All those types were added to DEncode. This mechanism reduces the amount of SSL handshakes and allows asynchronous requests from and to the client.

Developers are able to define a request timeout to specify a maximum time an RPC query or a file transfer can take. But this mechanism is not suitable for the new stable connections that can stay connected for a long period. To monitor the connection a keep alive mechanism has also been added to DISET. Periodically a packet is sent from the client to the server and back to verify that the connection is still working and that both ends still respond. In case the other end does not send the keep alive signal or doesn’t respond to it, DISET will raise a notification so the DIRAC component can react to the disconnect. This mechanism complements the global timeout DISET provides for RPC and file transfer capabilities.
4. Job optimizers

The process we had in mind when writing this new component is the job optimization. Before a job can be executed in an resource DIRAC has to prepare that job. This job preparation has different steps depending on the job requirements. In DIRAC the job preparation is done by the optimization chain. Up until now DIRAC’s optimization chain was built using Agents. Each Agent processed a different step in the optimization chain. Each Agent kept looking in the database for jobs that it could prepare. Once it found a job to prepare, it processed it and wrote the job back to the database. Then the next Agent would take over, load the job from the database, process it and store it back. This process was done by each Agent until the job had gone through all the optimization steps.

The Agent implementation of the optimization chain required:

- During each optimization step the Agent had to continuously poll the database.
- Slow reaction time. Even if there were many jobs ready to be prepared, they had to wait until the next optimization Agent queried the database. To minimize this effect, each Agent requested up to one thousand jobs to be prepared in each database query. This way the Agent would not do anything between database requests but if there were jobs ready it would prepare many of them without having to query the database again.
- Only one instance of each optimization Agent could be running at a single time. As each optimization Agent was requesting more than one job in the same request it was not possible to avoid two instances of the same optimization to take the same job without locking the whole database table.

To overcome the previously described problems, the optimization chain has been implemented using the Executor framework as seen on figure 3. The Optimization Mind can discover jobs to be processed either by querying the database or because another component notifies it of a job that needs to be prepared. In either case the Optimization Mind will load the job from the database and start the optimization chain for that job. The job information is serialized and sent to the first optimization Executor in the chain. The Executor will process the job, update the job information, serialize the result and send it back to the Optimization Mind. Executors do not write to the database, they modify the job state object, job manifest.. these objects will be serialized and sent back to the Optimization Mind. When the Optimization Mind receives the
data, it checks that the data received from the Executor can be properly deserialized and that it is consistent with the current state of the job in the database. Someone might have modified the job meanwhile it was being processed by an Executor, so the Executor sends the state of the job before and after optimization. If the initial state is the same as in the database, the modifications are then written to the database. Once the modifications have been committed, it will send the job to the next optimization Executor.

Each optimization step is coded as a task type in the Optimization Mind and each optimization Executor provides a list of steps that it knows how to process as task types. This process will be repeated until the job optimization is complete or until there’s an error in the job optimization chain. In either case the Optimization Mind will just set the appropriate job status in the database and delete the job from its memory. If there has been an Optimization Executor error such as an exception or disconnect, the Optimization Mind will just send the job to another instance of the same Optimization Executor if there’s any available. If there aren’t any more instances of the same Optimization Executor, the job will be placed in a queue waiting for one to be available.

A Executor can run more than one optimization step. In that case the Executor will try to process as many optimization steps as possible. For instance, if a job has 4 steps in the optimization chain and an Executor can execute the first three, it will do so and send back the result of the optimization after the first three steps to the Optimization Mind. This allows to minimize the amount of times a job has to go back to the Optimization Mind and thus also minimize database writes and network load.

If an optimization step fails, the job will be either declared as Failed and wait for the user or an operator to look at it, or it can be rescheduled to be retried again later. In the case of retrying a job, there is a maximum number of reschedules allowed. If the job reaches the maximum, it will be declared as Failed.

Using the Executor framework for the job optimization chain provides several benefits:

• Database load has been reduced. Jobs are loaded only once instead of having each Agent load the job independently.

• Several instances of each Optimization Executor can be started to increase the job processing throughput at any time.

• Having a non-responsive Optimization Executor doesn’t stop the job optimization chain.

5. Summary and outlook
The new Executor framework has been included in DIRAC. This new framework provides a new way of completing tasks is a more efficient way. The database load is reduced by having a single component that requires write access to it. Executors can be added or removed to adapt to the task load easily. Executors that get stalled or stop working don’t stop the system if there is more than one instance of them running. The Mind is still a single point of failure. Further developments may include adding redundancy to the Mind to increase the system resilience. In cases where Executors can process more than one task, it is worth trying out whether if it is more efficient to run Executors that can process one task type, more than one or a combination of them.

New stable connection and asynchronous request mechanisms have been added to DISET. The Executor framework relies on these new mechanisms to efficiently deliver tasks from the Mind to the Executors and back. To monitor the new connections a keep alive signaling has also been added to DISET. If a connection drops or gets stalled DISET will discover the failure and report it back to the component so it can properly react.

The DIRAC optimization chain implementation using the Executor framework is currently in the testing stage. It is functional but no performance tests have been done yet. There are two
DIRAC installations that are using this implementation that are being used to test the stability of the system.

Although DIRAC optimization chain has been implemented using this framework, several more DIRAC components will also be migrated to take advantage of the new features and benefits it provides.

Acknowledgments
The presented work has been financed by Comisión Interministerial de Ciencia y Tecnología (CICYT) (project FPA2010-21885-C02-01 and CPAN CSD2007-00042 from Programa Consolider-Ingenio 2010), and by Generalitat de Catalunya (AGAUR 2009SGR01268).

References
[1] Bencode. URL: http://en.wikipedia.org/wiki/Bencode.
[2] A. Casajus and R. Graciani. “DIRAC Distributed Secure Framework”. In: Computing in High-Energy Physics and Nuclear Physics 2009. 2009.
[3] A. Casajus and R. Graciani. “DIRAC Services and Agents”. In: Computing in High-Energy Physics and Nuclear Physics 2007. 2007.
[4] Remote procedure call. URL: http://en.wikipedia.org/wiki/Remote\_procedure\_call.
[5] Transport Layer Security. URL: http://en.wikipedia.org/wiki/Transport_Layer_Security.
[6] A. Tsaregorodtsev et al. “DIRAC: A community grid solution”. In: Computing in High-Energy Physics and Nuclear Physics 2007. 2008.