funcX

Release 0.0.1a5

May 22, 2020

Contents:

funcX is a High performance function serving system designed to orchestrate scientific workloads across heterogeneous computing resources, from laptops and local servers, through to campus clusters, clouds, and supercomputers.

CHAPTER 1

Quickstart

funcX is currently in Alpha and early testing releases are available on PyPI.

The latest version available on PyPI is `v0.0.1a4`.

You can try funcX on Binder

1.1 Installation

funcX comes with two components: the **endpoint agent** which is a user-launched services that make computation resources accessible for function executions, and the **funcX client** that enables the registration, execution and tracking of functions across **endpoints**.

Here are some pre-requisites for both the *endpoints* and the *funcX client*

1. Python3.6+
2. The machine must have outbound network access

To check if you have the right Python version, run the following commands:

```
>>> python3 --version
```

This should return the Python version, for eg: `Python 3.6.7`. Please note that that only the first two version numbers need to match.

To check if you have network access, run

```
>>> curl http://dev.funcx.org/api/v1/version
```

This should return a version string, for eg: `"0.0.1"`

1.1.1 Installation using Pip

While `pip` and `pip3` can be used to install funcX we suggest the following approach for reliable installation when many Python environments are avaialble.

1. Install funcX:

```
$ python3 -m pip install funcx
```

To update a previously installed funcX to a newer version, use: `python3 -m pip install -U funcx`

Note: The endpoint and client must use the same Python version. This is due to serialization differences between versions.

---

2. Install Jupyter for Tutorial notebooks:

```
$ python3 -m pip install jupyter
```

Note: For more detailed info on setting up Jupyter with Python3.5 go here

---

CHAPTER 2

Endpoints

An endpoint is a persistent service launched by the user on their compute system that serves as a conduit for routing and executing functions to their compute system. This could be their laptop, the login node of a campus cluster, grid, or supercomputing facility.

The endpoint can be configured to connect to the funcX webservice at funcx.org. Once the endpoint is registered you can invoke functions to be executed on it.

2.1 First time setup

For general use, running the initialize command via the *funcx-endpoint* CLI tool is the quickest way to setup your installation:

```
$ funcx-endpoint init
```

Initiating funcX will ask you to authenticate with Globus Auth. We require authentication in order to associate endpoints with users and enforce authentication and access control on the endpoint. As part of this step we request access to your identity information (to retrieve your email address) and Globus Groups management. We use Groups information to facilitate sharing of functions and endpoints by checking the Group membership of a group associated with a function.

Once you've run this command, a directory will be created at *$HOME/.funcx* and a set of configuration files will be generated. A default endpoint profile is also created that will be used, whenever you do not explicitly specify a profile to use for endpoint actions.

2.2 Configuring funcX

A configuration file will be created at *$HOME/.funcx/config.py*. This contains base information regarding the endpoint, such as a username and email. By default, this includes an address and port for a local broker, which is used if a local broker is deployed. You can also set the address of the endpoint for your workers to connect to. This is necessary if

your workers are not deployed on the same resource as the endpoint (e.g., when using a batch submission system, or cloud workers).

Note: If your funcX workers are not deployed on the same resource as the endpoint you must set the endpoint address for the workers to find the endpoint. This is done by setting the endpoint_address.

For example

```
import getpass
from parsl.addresses import address_by_route, address_by_hostname

global_options = {
  'username': getpass.getuser(),
  'email': 'USER@USERDOMAIN.COM',
  'broker_address': '127.0.0.1',
  'broker_port': 8088,
  'endpoint_address': address_by_hostname(),
}
```

2.3 Configuring an Endpoint

FuncX endpoints are designed to act as gateways to computational resources such as Clusters, Clouds, Supercomputers, and even your laptop. To make the best use of your resources, the endpoint must be configured to match the resources' capabilities and reflect the needs of the workloads you plan to execute. For example, you may want to limit the number of cores available to your endpointt.

FuncX provides you a rich class based configuration model that allows you to specify the shape of the resources (# of nodes, # of cores per worker, walltime etc) as well as place limits on how funcX may scale the resources in response to changing workload demands.

To generate the appropriate directories and default config template, run the following command:

```
$ funcx-endpoint configure <ENDPOINT_NAME>
```

The above command will create a profile for your endpoint in *$HOME/.funcx/<ENDPOINT_NAME>/* and will instantiate a *config.py* file. This file should be updated with the appropriate configurations for the computational system you are targeting before you start the endpoint. The funcX builds on Parsl and is configured using a `Config` object. For more information, see the `Config` class documentation.

Note: If the ENDPOINT_NAME is not specified, a default endpoint named "default" is configured.

2.4 Starting an Endpoint

To start a new endpoint run the following command:

```
$ funcx-endpoint start <ENDPOINT_NAME>
```

The above command will create a profile for your endpoint in *$HOME/.funcx/<ENDPOINT_NAME>/config.py*. This file should be updated with the appropriate configurations for the computational system you are targeting before you start the endpoint. To launch the endpoint, simply rerun the above command.

Note: If the ENDPOINT_NAME is not specified, a default endpoint named "default" is started.

Starting an endpoint will perform a registration process with the funcX Web Service. The registration process provides funcX with information regarding the endpoint. The Web Service then creates a Forwarder process for the endpoint and returns a UUID and connection information to the Forwarder. The endpoint will use this connection information to connect to the Forwarder. The endpoint establishes three outbound ZeroMQ channels to the forwarder (on the three ports returned during registration) to retrieve tasks, send results, and communicate command information.

Once started, the endpoint uses a daemon process to run in the background.

Warning: Only the owner of an endpoint is authorized to start an endpoint. Thus if you register with a different Globus Auth identity and try to start an endpoint owned by another identity, it will fail.

2.5 Stopping an Endpoint

To stop an endpoint, run the following command:

```
$ funcx-endpoint stop <ENDPOINT_NAME>
```

Note: If the ENDPOINT_NAME is not specified, the default endpoint is stopped.

Warning: Run the `funcx-endpoint stop` command **twice** to ensure that the endpoint is shutdown.

2.6 Listing Endpoints

To list available endpoints on the current system, run:

```
$ funcx-endpoint list
+---------------+-------------+--------------------------------------+
| Endpoint Name |   Status    |             Endpoint ID              |
+===============+=============+======================================+
| default       | Active      | 1e999502-b434-49a2-a2e0-d925383d2dd4 |
+---------------+-------------+--------------------------------------+
| KNL_test      | Inactive    | 8c01d13c-cfc1-42d9-96d2-52c51784ea16 |
+---------------+-------------+--------------------------------------+
| gpu_cluster   | Initialized | None                                 |
+---------------+-------------+--------------------------------------+
```

Endpoints can be the following states:

- **Initialized**: This status means that the endpoint has been created, but not started following configuration and not registered with the *funcx service*
- **Active**: This status means that the endpoint is active and available for executing functions
- **Inactive**: This status means that endpoint is not running right now and therefore, cannot service any functions.

CHAPTER 3

funcX Tutorial

funcX is a Function-as-a-Service (FaaS) platform for science that enables you to convert almost any computing resource into a high-performance function serving device. To do this, you deploy a funcX endpoint agent on the resource, which integrates it into the function serving fabric, allowing you to dynamically send, monitor, and receive results from function invocations. funcX is built on top of Parsl, enabling a funcX endpoint to use large compute resources via traditional batch queues, where funcX will dynamically provision, use, and release resources on-demand to fulfill function requests. The function service fabric, which is run centrally as a service, is hosted in AWS.

Here we provide an example of using funcX to register a function and run it on a publicly available tutorial endpoint.

3.1 funcX Client

We start by instantiating a funcX client as a programmatic means of communicating with the function service fabric. The client allows you to: - Register functions - Register containers and execution environments - Launch registered functions against endpoints - Check the status of launched functions - Retrieve outputs from functions

3.1.1 Authentication

Instantiating a client will force an authentication flow where you will be asked to authenticate with Globus Auth. Every interaction with funcX is authenticated to allow us to enforce access control on both functions and endpoints. As part of the authentication process we request access to your identity information (to retrieve your email address) and Globus Groups management access. We require Groups access in order to facilitate sharing.

```
[ ]: from funcx.sdk.client import FuncXClient

     fxc = FuncXClient()
```

Next we define a Python function, which we will later register with funcX. This function simply sums its input.

When defining a function you can specify *args and **kwargs as inputs.

Note: any dependencies for a funcX function must be specified inside the function body.

```
[ ]: def funcx_sum(items):
         return sum(items)
```

3.2 Registering a function

To use a function with funcX, you must first register it with the service, using `register_function`. You can optionally include a description of the function.

The registration process will serialize the function body and transmit it to the funcX function service fabric.

Registering a function returns a UUID for the function, which can then be used to invoke it.

```
[ ]: func_uuid = fxc.register_function(funcx_sum,
                                       description="A summation function")
     print(func_uuid)
```

3.3 Running a function

To invoke (perform) a function, you must provide the function's UUID, returned from the registration process, and an `endpoint_id`. Note: here we use the funcX public tutorial endpoint, which is running on AWS.

The client's `run` function will serialize any *args and **kwargs, and pass them to the function when invoking it. Therefore, as our example function simply takes an arg input (items), we can specify an input arg and it will be used by the function. Here we define a small list of integers for our function to sum.

The Web service will return the UUID for the invokation of the function, which we call a task. This UUID can be used to check the status of the task and retrieve the result.

```
[ ]: endpoint_uuid = '4b116d3c-1703-4f8f-9f6f-39921e5864df' # Public tutorial endpoint

     items = [1, 2, 3, 4, 5]

     res = fxc.run(items, endpoint_id=endpoint_uuid, function_id=func_uuid)
     print(res)
```

You can now retrieve the result of the invocation using `get_result()` on the UUID of the task.

```
[ ]: fxc.get_result(res)
```

CHAPTER 4

funcX Client

The **funcX Client** is the programmatic interface to FuncX from Python. The client provides a simple and intuitive interface to:

1. Register functions
2. Register containers and execution environments
3. Launch registered function against endpoints
4. Check the status of launched functions and
5. Retrieve outputs from functions

The following shows an example of creating a client.

```
from funcx.sdk.client import FuncXClient
fxc = FuncXClient()
```

Instantiating a client will start an authentication process where you will be asked to authenticate with Globus Auth. We require every interaction with the funcX Web Service to be authenticated using a Bearer token, this allows funcX to enforce access control on both functions and endpoints. As part of the authentication process we request access to your identity information (to retrieve your email address) and Globus Groups management access. We require Groups access in order to facilitate sharing. Users can share functions with others by associating a Globus Group with the function. If a group is set, the funcX Web Service uses Groups Management access to determine whether a user is a member of the specified group.

Note: The funcX Web Service internally caches function, endpoint, and access control lookups. Caches are based on user authentication tokens. To refresh the caches you can re-authenticate your client with *force_login=True*.

4.1 Registering Functions

You can register a Python function with funcX via *register_function()*. Function registration will serialize the function body and transmit it to the funcX Web Service. Once a function is registered with the service it will return a UUID that can be used to invoke the function.

Note: You must import any dependencies required by the function inside the function body.

You can associate a Globus Group with a function to enable sharing. If set, the Web Service will check if the invoking user is a member of the group. This is achieved by setting *group=<globus_group_id>* when registering a function.

You can also set a function to be publicly accssible by setting *public=True* when registering the function.

4.2 Running Functions

You can invoke a function using the UUID returned when registering the function. The client's *run()* function requires you to specify the *function_id* and *endpoint_id*. In addition, you can pass *args and **kwargs to the run function and they will be used when invoking your function. We serialize all inputs and outputs when running a function.

The result of your function's invocation can be retrieved using the client's *get_result()* function. This will either raise an exception if the task is still pending, or give you back the deserialized result of your invocation.

Note: If your function's execution raises an exception, get_result() will reraise it.

To minimize the overhead of communicating with the funcX Web Service we provide batch request and status capabilities. Documentation on batch requests can be found below.

4.3 Client Throttling

In order to avoid accidentally DoS'ing the funcX Web Service we place soft throttling restrictions on the funcX client. There are two key throttling measures: firstly, we limit the number of requests a client can make to the Web Service to 5 every 5 seconds, and secondly, we limit the size of inputs and outputs transmitted through the service to 2MB.

Batching requests and status can help reduce the number of requests made to the Web Service. In addition, the limit on the number of requests made to the Web Service can be removed by setting *throttling_enabled* to False.

```
fxc = FuncXClient()
fxc.throttling_enabled = False
```

4.4 FuncXClient Reference:

class funcx.**FuncXClient**(*http_timeout=None*, *funcx_home='~/.funcx'*, *force_login=False*, *fx_authorizer=None*, *funcx_service_address='https://dev.funcx.org/api/v1'*, ***kwargs*)

Main class for interacting with the funcX service

Holds helper operations for performing common tasks with the funcX service.

add_to_whitelist(*endpoint_id*, *function_ids*)
Adds the function to the endpoint's whitelist

Parameters

- **endpoint_id** (*str*) – The uuid of the endpoint
- **function_ids** (*list*) – A list of function id's to be whitelisted

Returns The response of the request

Return type json

batch_run(*batch*)
Initiate a batch of tasks to funcX

Parameters **batch** (*a Batch object*) –

Returns task_ids

Return type a list of UUID strings that identify the tasks

create_batch()
Create a Batch instance to handle batch submission in funcX

Returns Status block containing "status" key.

Return type Batch instance

delete_from_whitelist(*endpoint_id*, *function_ids*)
List the endpoint's whitelist

Parameters

- **endpoint_id** (*str*) – The uuid of the endpoint
- **function_ids** (*list*) – A list of function id's to be whitelisted

Returns The response of the request

Return type json

get_batch_result(*task_id_list*)
Request results for a batch of task_ids

get_batch_status(*task_id_list*)
Request status for a batch of task_ids

get_container(*container_uuid*, *container_type*)
Get the details of a container for staging it locally.

Parameters

- **container_uuid** (*str*) – UUID of the container in question
- **container_type** (*str*) – The type of containers that will be used (Singularity, Shifter, Docker)

Returns The details of the containers to deploy

Return type dict

get_containers(*name*, *description=None*)
Register a DLHub endpoint with the funcX service and get the containers to launch.

Parameters

- **name** (*str*) – Name of the endpoint

- **description** (*str*) – Description of the endpoint

Returns The port to connect to and a list of containers

Return type int

get_endpoint_status(*endpoint_uuid*)
Get the status reports for an endpoint.

Parameters **endpoint_uuid** (*str*) – UUID of the endpoint in question

Returns The details of the endpoint's stats

Return type dict

get_result(*task_id*)
Get the result of a funcX task

Parameters **task_id** (*str*) – UUID of the task

Returns **Result obj**

Return type If task completed

Raises Exception obj: Exception due to which the task failed

get_task_status(*task_id*)
Get the status of a funcX task.

Parameters **task_id** (*str*) – UUID of the task

Returns Status block containing "status" key.

Return type dict

get_whitelist(*endpoint_id*)
List the endpoint's whitelist

Parameters **endpoint_id** (*str*) – The uuid of the endpoint

Returns The response of the request

Return type json

logout()
Remove credentials from your local system

map_run(**args*, *endpoint_id=None*, *function_id=None*, *asynchronous=False*, ***kwargs*)
Initiate an invocation

Parameters

- ***args** (*Any*) – Args as specified by the function signature
- **endpoint_id** (*uuid str*) – Endpoint UUID string. Required
- **function_id** (*uuid str*) – Function UUID string. Required
- **asynchronous** (*bool*) – Whether or not to run the function asynchronously

Returns

- **task_id** (*str*)
- *UUID string that identifies the task*

register_container(*location*, *container_type*, *name=''*, *description=''*)
Register a container with the funcX service.

Parameters

- **location** (*str*) – The location of the container (e.g., its docker url). Required
- **container_type** (*str*) – The type of containers that will be used (Singularity, Shifter, Docker). Required
- **name** (*str*) – A name for the container. Default = ''
- **description** (*str*) – A description to associate with the container. Default = ''

Returns The id of the container

Return type str

register_endpoint(*name*, *endpoint_uuid*, *description=None*)
Register an endpoint with the funcX service.

Parameters

- **name** (*str*) – Name of the endpoint
- **endpoint_uuid** (*str*) – The uuid of the endpoint
- **description** (*str*) – Description of the endpoint

Returns

{'endopoint_id' [<>,] 'address' : <>, 'client_ports': <>}

Return type A dict

register_function(*function*, *function_name=None*, *container_uuid=None*, *description=None*, *public=False*, *group=None*)
Register a function code with the funcX service.

Parameters

- **function** (*Python Function*) – The function to be registered for remote execution
- **function_name** (*str*) – The entry point (function name) of the function. Default: None
- **container_uuid** (*str*) – Container UUID from registration with funcX
- **description** (*str*) – Description of the file
- **public** (*bool*) – Whether or not the function is publicly accessible. Default = False
- **group** (*str*) – A globus group uuid to share this function with

Returns **function uuid** – UUID identifier for the registered function

Return type str

run(**args*, *endpoint_id=None*, *function_id=None*, ***kwargs*)
Initiate an invocation

Parameters

- ***args** (*Any*) – Args as specified by the function signature
- **endpoint_id** (*uuid str*) – Endpoint UUID string. Required
- **function_id** (*uuid str*) – Function UUID string. Required
- **asynchronous** (*bool*) – Whether or not to run the function asynchronously

Returns

- **task_id** (*str*)
- *UUID string that identifies the task*

update_table(*return_msg*, *task_id*)

Parses the return message from the service and updates the internal func_tables

Parameters

- **return_msg** (*str*) – Return message received from the funcx service
- **task_id** (*str*) – task id string

CHAPTER 5

Reference

funcx	funcX : Fast function serving for clouds, clusters and supercomputers.
funcx.sdk	
funcx.sdk.utils	
funcx.serialize	
funcx.endpoint	
funcx.executors	
funcx.providers	
funcx.strategies	
funcx.queues	
funcx.mock_broker	

5.1 funcx package

5.1.1 Subpackages

funcx.endpoint package

Subpackages

funcx.endpoint.utils package

Submodules

funcx.endpoint.utils.MDP module

Majordomo Protocol definitions

funcx.endpoint.utils.zhelpers module

Helper module for example applications. Mimics ZeroMQ Guide's zhelpers.h.

funcx.endpoint.utils.zhelpers.**dump**(*msg_or_socket*)
Receives all message parts from socket, printing each frame neatly

funcx.endpoint.utils.zhelpers.**set_id**(*zsocket*)
Set simple random printable identity on socket

funcx.endpoint.utils.zhelpers.**socket_set_hwm**(*socket*, *hwm=-1*)
libzmq 2/3/4 compatible sethwm

funcx.endpoint.utils.zhelpers.**zpipe**(*ctx*)
build inproc pipe for talking to threads mimic pipe used in czmq zthread_fork. Returns a pair of PAIRs connected via inproc

funcx.endpoint.utils.zmq_worker module

class funcx.endpoint.utils.zmq_worker.**ZMQWorker**(*broker*, *service*, *verbose=False*)
Bases: object

Majordomo Protocol Worker API, Python version

Implements the MDP/Worker spec at http:#rfc.zeromq.org/spec:7.

HEARTBEAT_LIVENESS = 3

broker = None

ctx = None

destroy()

expect_reply = False

heartbeat = 2500

heartbeat_at = 0

liveness = 0

reconnect = 2500

reconnect_to_broker()
Connect or reconnect to broker

recv()
Send reply, if any, to broker and wait for next request.

reply_to = None

send(*reply*, *reply_to*)
Send a reply to the broker.

send_to_broker(*command*, *option=None*, *msg=None*)
Send message to broker.

If no msg is provided, creates one internally

service = None

timeout = 2500

verbose = False

worker = None

Module contents

Submodules

funcx.endpoint.auth module

funcx.endpoint.auth.**load_auth_client**()
Create an AuthClient for the portal

No credentials are used if the server is not production

Returns Client used to perform GlobusAuth actions

Return type globus_sdk.ConfidentialAppAuthClient

funcx.endpoint.config module

funcx.endpoint.endpoint module

funcx.endpoint.endpoint.**check_pidfile**(*filepath*, *match_name*, *endpoint_name*)
Helper function to identify possible dead endpoints

funcx.endpoint.endpoint.**cli_run**()
Entry point for funcx-endpoint

funcx.endpoint.endpoint.**configure_endpoint**(*args*, *config_file=None*, *global_config=None*)
Configure an endpoint

Drops a config.py template into the funcx configs directory. The template usually goes to ~/.funcx/<ENDPOINT_NAME>/config.py

Parameters

- **args** (*args object*) – Args object from the arg parsing
- **config_file** (*str*) – Path to a config file to be used instead of the funcX default config file
- **global_config** (*dict*) – Global config dict

funcx.endpoint.endpoint.**init_endpoint**(*args*)
Setup funcx dirs and config files including a default endpoint config

TODO : Every mechanism that will update the config file, must be using a locking mechanism, ideally something like fcntl https://docs.python.org/3/library/fcntl.html to ensure that multiple endpoint invocations do not mangle the funcx config files or the lockfile module.

funcx.endpoint.endpoint.**init_endpoint_dir**(*funcx_dir*, *endpoint_name*, *config_file=None*)
Initialize a clean endpoint dir

Returns if an endpoint_dir already exists

Parameters

- **funcx_dir** (*str*) – Path to the funcx_dir on the system

- **endpoint_name** (*str*) – Name of the endpoint, which will be used to name the dir for the endpoint.
- **config_file** (*str*) – Path to a config file to be used instead of the funcX default config file

funcx.endpoint.endpoint.**load_endpoint**(*endpoint_dir*)

Parameters **endpoint_dir** (*str*) – endpoint directory path within funcx_dir

funcx.endpoint.endpoint.**register_endpoint**(*funcx_client*, *endpoint_name*, *endpoint_uuid*, *endpoint_dir*)

Register the endpoint and return the registration info.

Parameters

- **funcx_client** (FuncXClient) – The auth'd client to communicate with the funcX service
- **endpoint_name** (*str*) – The name to register the endpoint with
- **endpoint_uuid** (*str*) – The uuid to register the endpoint with
- **endpoint_dir** (*str*) – The directory to write endpoint registration info into.

funcx.endpoint.endpoint.**register_with_hub**(*endpoint_uuid*, *endpoint_dir*, *address*, *redis_host='funcx-redis.wtgh6h.0001.use1.cache.amazonaws.com'*)

This currently registers directly with the Forwarder micro service.

Can be used as an example of how to make calls this it, while the main API is updated to do this calling on behalf of the endpoint in the second iteration.

funcx.endpoint.endpoint.**start_endpoint**(*args*, *global_config=None*)

Start an endpoint

This function will do: 1. Connect to the broker service, and register itself 2. Get connection info from broker service 3. Start the interchange as a daemon

Broker service |
—2—> Forwarder |
/register <—3—+ ^ |

+—^————————+———-+

| |

1 4 6 | v |

+—+—+—+ v | Start |—5—> Interchange | Endpoint | daemon +————————+

Parameters

- **args** (*args object*) – Args object from the arg parsing
- **global_config** (*dict*) – Global config dict

funcx.endpoint.endpoint.**stop_endpoint**(*args*, *global_config=None*)

Stops an endpoint using the pidfile

Parameters

- **args** –

- **global_config** –

funcx.endpoint.list_endpoints module

funcx.endpoint.list_endpoints.**list_endpoints**(*args*)
List all available endpoints

funcx.endpoint.version module

Module contents

funcx.executors package

Subpackages

funcx.executors.high_throughput package

Submodules

funcx.executors.high_throughput.container_sched module

funcx.executors.high_throughput.container_sched.**naive_scheduler**(*task_qs*, *outstanding_task_count*, *max_workers*, *old_worker_map*, *to_die_list*, *logger*)

Return two items (as one tuple) dict kill_list :: KILL [(worker_type, num_kill), . . .] dict create_list :: CREATE [(worker_type, num_create), . . .]

In this scheduler model, there is minimum 1 instance of each nonempty task queue.

funcx.executors.high_throughput.default_config module

funcx.executors.high_throughput.executor module

HighThroughputExecutor builds on the Swift/T EMEWS architecture to use MPI for fast task distribution

There's a slow but sure deviation from Parsl's Executor interface here, that needs to be addressed.

class funcx.executors.high_throughput.executor.**HighThroughputExecutor**(*label='HighThroughputExecu*
provider=LocalProvider(
chan-
nel=LocalChannel(
envs={},
script_dir=None,
user-
home='/home/docs/checkouts/
),
cmd_timeout=30,
init_blocks=4,
launcher=SingleNodeLaunche
max_blocks=10,
min_blocks=0,
move_files=None,
nodes_per_block=1,
par-
al-
lelism=1,
wall-
time='00:15:00',
worker_init='
),
launch_cmd=None,
ad-
dress='127.0.0.1',
worker_ports=None,
worker_port_range=(54000,
55000),
inter-
change_port_range=(55000,
56000),
stor-
age_access=None,
work-
ing_dir=None,
worker_debug=False,
cores_per_worker=1.0,
max_workers=inf,
heart-
beat_threshold=120,
heart-
beat_period=30,
poll_period=10,
con-
tainer_image=None,
worker_mode='singularity_re
sup-
press_failure=False,
end-
point_id=None,
end-
point_db=None,
man-
aged=True)

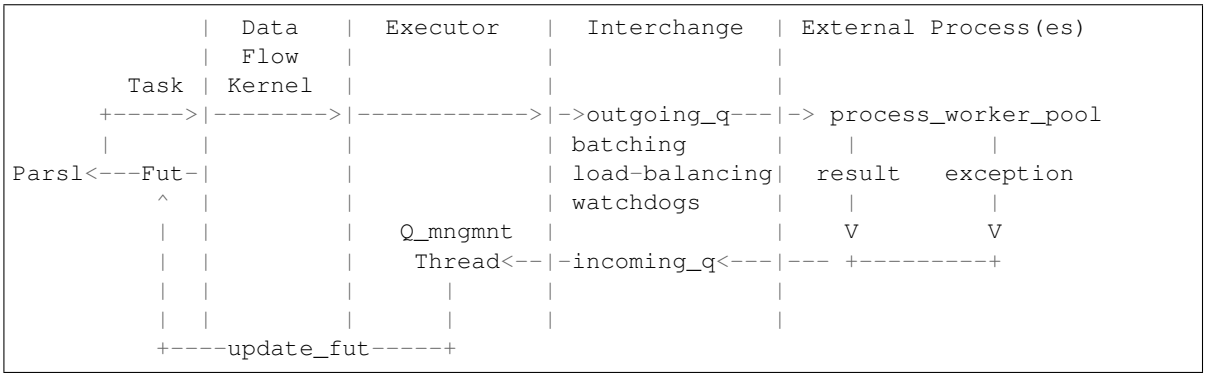
Bases: `parsl.executors.base.ParslExecutor`, `parsl.utils.RepresentationMixin`

Executor designed for cluster-scale

The HighThroughputExecutor system has the following components:

1. The HighThroughputExecutor instance which is run as part of the Parsl script.
2. The Interchange which is acts as a load-balancing proxy between workers and Parsl
3. The multiprocessing based worker pool which coordinates task execution over several cores on a node.
4. ZeroMQ pipes connect the HighThroughputExecutor, Interchange and the process_worker_pool

Here is a diagram



Parameters

- **provider** (`ExecutionProvider`) –

  Provider to access computation resources. Can be one of `EC2Provider`, `Cobalt`, `Condor`, `GoogleCloud`, `GridEngine`, `Jetstream`, `Local`, `GridEngine`, `Slurm`, or `Torque`.

- **label** (*str*) – Label for this executor instance.

- **launch_cmd** (*str*) – Command line string to launch the process_worker_pool from the provider. The command line string will be formatted with appropriate values for the following values (debug, task_url, result_url, cores_per_worker, nodes_per_block, heartbeat_period ,heartbeat_threshold, logdir). For eg: launch_cmd="process_worker_pool.py {debug} -c {cores_per_worker} –task_url={task_url} –result_url={result_url}"

- **address** (*string*) – An address to connect to the main Parsl process which is reachable from the network in which workers will be running. This can be either a hostname as returned by *hostname* or an IP address. Most login nodes on clusters have several network interfaces available, only some of which can be reached from the compute nodes. Some trial and error might be necessary to indentify what addresses are reachable from compute nodes.

- **worker_ports** (*(int, int)*) – Specify the ports to be used by workers to connect to Parsl. If this option is specified, worker_port_range will not be honored.

- **worker_port_range** (*(int, int)*) – Worker ports will be chosen between the two integers provided.

- **interchange_port_range** (*(int, int)*) – Port range used by Parsl to communicate with the Interchange.

- **working_dir** (*str*) – Working dir to be used by the executor.

- **worker_debug** (*Bool*) – Enables worker debug logging.
- **managed** (*Bool*) – If this executor is managed by the DFK or externally handled.
- **cores_per_worker** (*float*) – cores to be assigned to each worker. Oversubscription is possible by setting cores_per_worker < 1.0. Default=1
- **max_workers** (*int*) – Caps the number of workers launched by the manager. Default: infinity
- **suppress_failure** (*Bool*) – If set, the interchange will suppress failures rather than terminate early. Default: False
- **heartbeat_threshold** (*int*) – Seconds since the last message from the counterpart in the communication pair: (interchange, manager) after which the counterpart is assumed to be un-available. Default:120s
- **heartbeat_period** (*int*) – Number of seconds after which a heartbeat message indicating liveness is sent to the counterpart (interchange, manager). Default:30s
- **poll_period** (*int*) – Timeout period to be used by the executor components in milliseconds. Increasing poll_periods trades performance for cpu efficiency. Default: 10ms
- **container_image** (*str*) – Path or identfier to the container image to be used by the workers
- **= None** (*endpoint_db*) – Endpoint DB object
- **worker_mode** (*str*) – Select the mode of operation from no_container, singularity_reuse, singularity_single_use Default: singularity_reuse

connected_workers

connection_info
All connection info necessary for the endpoint to connect back

Returns Dict with connection info

hold_worker(*worker_id*)
Puts a worker on hold, preventing scheduling of additional tasks to it.

This is called "hold" mostly because this only stops scheduling of tasks, and does not actually kill the worker.

Parameters **worker_id** (*str*) – Worker id to be put on hold

initialize_scaling()
Compose the launch command and call the scale_out

This should be implemented in the child classes to take care of executor specific oddities.

outstanding

request_status_info()

scale_in(*blocks*)
Scale in the number of active blocks by specified amount.

The scale in method here is very rude. It doesn't give the workers the opportunity to finish current tasks or cleanup. This is tracked in issue #530

Raises NotImplementedError

scale_out(*blocks=1*)
Scales out the number of blocks by "blocks"

Raises NotImplementedError

scaling_enabled
Specify if scaling is enabled.

The callers of ParslExecutors need to differentiate between Executors and Executors wrapped in a resource provider

shutdown(*hub=True*, *targets='all'*, *block=False*)
Shutdown the executor, including all workers and controllers.

This is not implemented.

Kwargs:

- hub (Bool): Whether the hub should be shutdown, Default:True,
- targets (list of ints| 'all'): List of block id's to kill, Default:'all'
- block (Bool): To block for confirmations or not

Raises NotImplementedError

start()
Create the Interchange process and connect to it.

status()
Return status of all blocks.

submit(*bufs*, *task_id=None*)
Submits work to the the outgoing_q.

The outgoing_q is an external process listens on this queue for new work. This method behaves like a submit call as described here Python docs:

Parameters

- **- Pickled buffer with (b'<Function>', b'<args>', b'<kwargs>')** (*Bufs*) –
- **Returns** – Future

wait_for_endpoint()

weakref_cb(*q=None*)
We do not use this yet.

funcx.executors.high_throughput.executor.**executor_starter**(*htex*, *logdir*, *endpoint_id*, *logging_level=10*)

funcx.executors.high_throughput.funcx_manager module

class funcx.executors.high_throughput.funcx_manager.**Manager**(*task_q_url='tcp://127.0.0.1:50097'*, *result_q_url='tcp://127.0.0.1:50098'*, *max_queue_size=10*, *cores_per_worker=1*, *max_workers=inf*, *uid=None*, *heartbeat_threshold=120*, *heartbeat_period=30*, *logdir=None*, *debug=False*, *block_id=None*, *internal_worker_port_range=(50000, 60000)*, *worker_mode='singularity_reuse'*, *scheduler_mode='hard'*, *worker_type=None*, *worker_max_idletime=60*, *poll_period=100*)

Bases: object

Manager manages task execution by the workers

0mq | Manager | Worker Processes
| |
<—Request N task—–+–Count task reqs | Request task<–+

Interchange | ————————-+->Receive task batch| | |

| Distribute tasks–+—> Get(block) & |
| | Execute task |
| | | |
<————————+–Return results—-+—- Post result |
| | | |
| | +———-+
| IPC-Qeueues

create_reg_message()
Creates a registration message to identify the worker to the interchange

heartbeat()
Send heartbeat to the incoming task queue

pull_tasks(*kill_event*)
Pull tasks from the incoming tasks 0mq pipe onto the internal pending task queue

While : receive results and task requests from the workers receive tasks/heartbeats from the Interchange match tasks to workers if task doesn't have appropriate worker type:

launch worker of type.. with LRU or some sort of caching strategy.

if workers >> tasks: advertize available capacity

kill_event [threading.Event] Event to let the thread know when it is time to die.

push_results(*kill_event*, *max_result_batch_size=1*)
Listens on the pending_result_queue and sends out results via 0mq

kill_event [threading.Event] Event to let the thread know when it is time to die.

remove_worker_init(*worker_type*)
Kill/Remove a worker of a given worker_type.

Add a kill message to the task_type queue.

Assumption : All workers of the same type are uniform, and therefore don't discriminate when killing.

start()

- **while True:** Receive tasks and start appropriate workers Push tasks to available workers Forward results

funcx.executors.high_throughput.funcx_manager.**cli_run**()

funcx.executors.high_throughput.funcx_worker module

class funcx.executors.high_throughput.funcx_worker.**FuncXWorker**(*worker_id*, *address*, *port*, *logdir*, *debug=False*, *worker_type='RAW'*)

Bases: object

The FuncX worker :param worker_id: Worker id string :type worker_id: str :param address: Address at which the manager might be reached. This is usually 127.0.0.1 :type address: str :param port: Port at which the manager can be reached :type port: int :param logdir: Logging directory :type logdir: str :param debug: Enables debug logging :type debug: Bool

Funcx worker will use the REP sockets to: task = recv () result = execute(task) send(result)

execute_task(*message*)
Deserialize the buffer and execute the task.

Returns the result or throws exception.

registration_message()

start()

funcx.executors.high_throughput.funcx_worker.**cli_run**()

funcx.executors.high_throughput.global_config module

funcx.executors.high_throughput.interchange module

exception funcx.executors.high_throughput.interchange.**BadRegistration**(*worker_id*, *critical=False*)

Bases: Exception

A new Manager tried to join the executor with a BadRegistration message

class funcx.executors.high_throughput.interchange.**Interchange**(*config*, *client_address='127.0.0.1'*, *interchange_address='127.0.0.1'*, *client_ports=(50055, 50056, 50057)*, *worker_ports=None*, *worker_port_range=(54000, 55000)*, *cores_per_worker=1.0*, *worker_debug=False*, *launch_cmd=None*, *heartbeat_threshold=60*, *logdir='.'*, *logging_level=20*, *poll_period=10*, *endpoint_id=None*, *suppress_failure=False*, *max_heartbeats_missed=2*)

Bases: object

Interchange is a task orchestrator for distributed systems.

1. Asynchronously queue large volume of tasks (>100K)
2. Allow for workers to join and leave the union
3. Detect workers that have failed using heartbeats
4. Service single and batch requests from workers
5. Be aware of requests worker resource capacity, eg. schedule only jobs that fit into walltime.

TODO: We most likely need a PUB channel to send out global commands, like shutdown

get_container(*container_uuid*)
Get the container image location if it is not known to the interchange

get_outstanding_breakdown()
Get outstanding breakdown per manager and in the interchange queues

Returns

- *List of status for online elements*
- *[(element, tasks_pending, status) ...]*

get_status_report()
Get utilization numbers

get_tasks(*count*)
Obtains a batch of tasks from the internal pending_task_queue

Parameters **count** (*int*) – Count of tasks to get from the queue

Returns eg. [{'task_id':<x>, 'buffer':<buf>} ...]

Return type List of upto count tasks. May return fewer than count down to an empty list

get_total_live_workers()
Get the total active workers

get_total_tasks_outstanding()
Get the outstanding tasks in total

hold_manager(*manager*)
Put manager on hold :param manager: Manager id to be put on hold while being killed :type manager: str

load_config()
Load the config

migrate_tasks_to_internal(*kill_event*, *status_request*)
Pull tasks from the incoming tasks 0mq pipe onto the internal pending task queue

kill_event [threading.Event] Event to let the thread know when it is time to die.

provider_status()
Get status of all blocks from the provider

scale_in(*blocks=None*, *block_ids=[]*, *task_type=None*)
Scale in the number of active blocks by specified amount.

Parameters

- **blocks** (*int*) – # of blocks to terminate
- **block_ids** (*[str..]*) – List of external block ids to terminate

scale_out(*blocks=1*, *task_type=None*)
Scales out the number of blocks by "blocks"

Raises NotImplementedError

start(*poll_period=None*)
Start the Interchange

poll_period [int] poll_period in milliseconds

stop()
Prepare the interchange for shutdown

exception funcx.executors.high_throughput.interchange.**ManagerLost**(*worker_id*)
Bases: Exception

Task lost due to worker loss. Worker is considered lost when multiple heartbeats have been missed.

exception funcx.executors.high_throughput.interchange.**ShutdownRequest**
Bases: Exception

Exception raised when any async component receives a ShutdownRequest

funcx.executors.high_throughput.interchange.**cli_run**()

funcx.executors.high_throughput.interchange.**start_file_logger**(*filename*, *name='interchange'*, *level=10*, *format_string=None*)

Add a stream log handler.

Parameters

- **filename** (*string*) – Name of the file to write logs to. Required.
- **name** (*string*) – Logger name. Default="parsl.executors.interchange"

- **level** (*logging.LEVEL*) – Set the logging level. Default=logging.DEBUG - format_string (string): Set the format string
- **format_string** (*string*) – Format string to use.

Returns

Return type None.

funcx.executors.high_throughput.interchange.**starter**(*comm_q*, **args*, ***kwargs*)
Start the interchange process

The executor is expected to call this function. The args, kwargs match that of the Interchange.__init__

funcx.executors.high_throughput.interchange_task_dispatch module

funcx.executors.high_throughput.interchange_task_dispatch.**dispatch**(*interesting_managers*, *pending_task_queue*, *ready_manager_queue*, *scheduler_mode='hard'*, *loop='first'*, *task_dispatch=None*, *dispatched_tasks=0*)
This is the core task dispatching algorithm for interchange. The algorithm depends on the scheduler mode and which loop.

funcx.executors.high_throughput.interchange_task_dispatch.**get_tasks_hard**(*pending_task_queue*, *manager_ads*, *real_capacity*)

funcx.executors.high_throughput.interchange_task_dispatch.**get_tasks_soft**(*pending_task_queue*, *manager_ads*, *real_capacity*, *loop='first'*)

funcx.executors.high_throughput.interchange_task_dispatch.**naive_interchange_task_dispatch**(*i*

This is an initial task dispatching algorithm for interchange. It returns a dictionary, whose key is manager, and the value is the list of tasks to be sent to manager, and the total number of dispatched tasks.

funcx.executors.high_throughput.worker_map module

class funcx.executors.high_throughput.worker_map.**WorkerMap**(*max_worker_count*)
Bases: object

WorkerMap keeps track of workers

add_worker(*worker_id='0.3343746937276747'*, *mode='no_container'*, *worker_type='RAW'*, *container_uri=None*, *walltime=1*, *address=None*, *debug=None*, *worker_port=None*, *logdir=None*, *uid=None*)
Launch the appropriate worker

Parameters

- **worker_id** (*str*) – Worker identifier string
- **mode** (*str*) – Valid options are no_container, singularity
- **walltime** (*int*) – Walltime in seconds before we check status

get_next_worker_q(*new_worker_map*)

Helper function to generate a queue of next workers to spin up . From a mapping generated by the scheduler

Parameters **new_worker_map** (*dict*) – {worker_type: total_number_of_containers,...}

Returns

Return type Queue containing the next workers the system should spin-up.

get_worker(*worker_type*)
Get a task and reduce the # of worker for that type by 1. Raises queue.Empty if empty

get_worker_counts()
Returns just the dict of worker_type and counts

put_worker(*worker*)
Adds worker to the list of waiting workers

ready_worker_count()

register_worker(*worker_id*, *worker_type*)
Add a new worker

remove_worker(*worker_id*)
Remove the worker from the WorkerMap

Should already be KILLed by this point.

spin_down_workers(*new_worker_map*, *worker_max_idletime=60*, *need_more=False*, *scheduler_mode='hard'*)
Helper function to call 'remove' for appropriate workers in 'new_worker_map'.

Parameters **new_worker_map** (*dict*) – {worker_type: total_number_of_containers,...}.

Returns

Return type List of removed worker types.

spin_up_workers(*next_worker_q*, *address=None*, *debug=None*, *uid=None*, *logdir=None*, *worker_port=None*)
Helper function to call 'remove' for appropriate workers in 'new_worker_map'.

Parameters

- **new_worker_q** (*queue.Queue()*) – Queue of worker types to be spun up next.
- **address** (*str*) – Address at which to connect to the workers.
- **debug** (*bool*) – Whether debug logging is activated.
- **uid** (*str*) – Worker ID to be assigned to worker.

- **logdir** (*str*) – Directory in which to write logs
- **worker_port** (*int*) – Port at which to connect to the workers.

Returns

Return type Total number of spun-up workers.

update_worker_idle(*worker_type*)
Update the workers' last idle time by worker type

funcx.executors.high_throughput.zmq_pipes module

class funcx.executors.high_throughput.zmq_pipes.**CommandClient**(*ip_address*, *port_range*)

Bases: object

close()

run(*message*)
This function needs to be fast at the same time aware of the possibility of ZMQ pipes overflowing.

The timeout increases slowly if contention is detected on ZMQ pipes. We could set copy=False and get slightly better latency but this results in ZMQ sockets reaching a broken state once there are ~10k tasks in flight. This issue can be magnified if each the serialized buffer itself is larger.

class funcx.executors.high_throughput.zmq_pipes.**ResultsIncoming**(*ip_address*, *port_range*)

Bases: object

Incoming results queue from the Interchange to the executor

close()

get(*block=True*, *timeout=None*)

request_close()

class funcx.executors.high_throughput.zmq_pipes.**TasksOutgoing**(*ip_address*, *port_range*)

Bases: object

Outgoing task queue from the executor to the Interchange

close()

put(*message*, *max_timeout=1000*)
This function needs to be fast at the same time aware of the possibility of ZMQ pipes overflowing.

The timeout increases slowly if contention is detected on ZMQ pipes. We could set copy=False and get slightly better latency but this results in ZMQ sockets reaching a broken state once there are ~10k tasks in flight. This issue can be magnified if each the serialized buffer itself is larger.

Parameters

- **message** (*py object*) – Python object to send
- **max_timeout** (*int*) – Max timeout in milliseconds that we will wait for before raising an exception

Raises zmq.EAGAIN if the send failed.

Module contents

Module contents

class funcx.executors.**HighThroughputExecutor**(*label='HighThroughputExecutor'*, *provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, user-home='/home/docs/checkouts/readthedocs.org/user_builds/funcx/ch), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init=''), launch_cmd=None, address='127.0.0.1', worker_ports=None, worker_port_range=(54000, 55000), interchange_port_range=(55000, 56000), storage_access=None, working_dir=None, worker_debug=False, cores_per_worker=1.0, max_workers=inf, heartbeat_threshold=120, heartbeat_period=30, poll_period=10, container_image=None, worker_mode='singularity_reuse', suppress_failure=False, endpoint_id=None, endpoint_db=None, managed=True*)

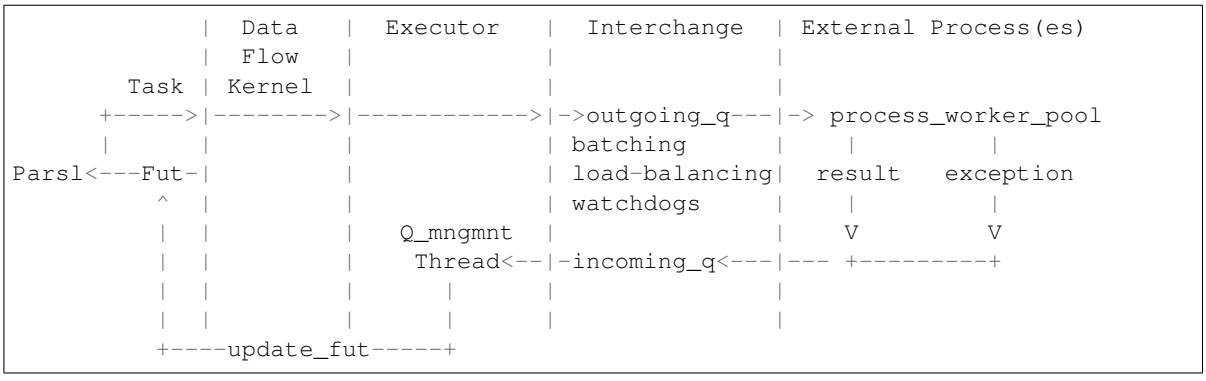
Bases: parsl.executors.base.ParslExecutor, parsl.utils.RepresentationMixin

Executor designed for cluster-scale

The HighThroughputExecutor system has the following components:

1. The HighThroughputExecutor instance which is run as part of the Parsl script.
2. The Interchange which is acts as a load-balancing proxy between workers and Parsl
3. The multiprocessing based worker pool which coordinates task execution over several cores on a node.
4. ZeroMQ pipes connect the HighThroughputExecutor, Interchange and the process_worker_pool

Here is a diagram



Parameters

- **provider** (ExecutionProvider) –

  Provider to access computation resources. Can be one of EC2Provider, Cobalt, Condor, GoogleCloud, GridEngine, Jetstream, Local, GridEngine, Slurm, or Torque.

- **label** (*str*) – Label for this executor instance.
- **launch_cmd** (*str*) – Command line string to launch the process_worker_pool from the provider. The command line string will be formatted with appropriate values for the following values (debug, task_url, result_url, cores_per_worker, nodes_per_block, heartbeat_period ,heartbeat_threshold, logdir). For eg: launch_cmd="process_worker_pool.py {debug} -c {cores_per_worker} –task_url={task_url} –result_url={result_url}"
- **address** (*string*) – An address to connect to the main Parsl process which is reachable from the network in which workers will be running. This can be either a hostname as returned by *hostname* or an IP address. Most login nodes on clusters have several network interfaces available, only some of which can be reached from the compute nodes. Some trial and error might be necessary to indentify what addresses are reachable from compute nodes.
- **worker_ports** (*(int, int)*) – Specify the ports to be used by workers to connect to Parsl. If this option is specified, worker_port_range will not be honored.
- **worker_port_range** (*(int, int)*) – Worker ports will be chosen between the two integers provided.
- **interchange_port_range** (*(int, int)*) – Port range used by Parsl to communicate with the Interchange.
- **working_dir** (*str*) – Working dir to be used by the executor.
- **worker_debug** (*Bool*) – Enables worker debug logging.
- **managed** (*Bool*) – If this executor is managed by the DFK or externally handled.
- **cores_per_worker** (*float*) – cores to be assigned to each worker. Oversubscription is possible by setting cores_per_worker < 1.0. Default=1
- **max_workers** (*int*) – Caps the number of workers launched by the manager. Default: infinity
- **suppress_failure** (*Bool*) – If set, the interchange will suppress failures rather than terminate early. Default: False
- **heartbeat_threshold** (*int*) – Seconds since the last message from the counterpart in the communication pair: (interchange, manager) after which the counterpart is assumed to be un-available. Default:120s
- **heartbeat_period** (*int*) – Number of seconds after which a heartbeat message indicating liveness is sent to the counterpart (interchange, manager). Default:30s
- **poll_period** (*int*) – Timeout period to be used by the executor components in milliseconds. Increasing poll_periods trades performance for cpu efficiency. Default: 10ms
- **container_image** (*str*) – Path or identfier to the container image to be used by the workers
- **= None** (*endpoint_db*) – Endpoint DB object
- **worker_mode** (*str*) – Select the mode of operation from no_container, singularity_reuse, singularity_single_use Default: singularity_reuse

connected_workers

connection_info
All connection info necessary for the endpoint to connect back

Returns Dict with connection info

hold_worker(*worker_id*)
Puts a worker on hold, preventing scheduling of additional tasks to it.

This is called "hold" mostly because this only stops scheduling of tasks, and does not actually kill the worker.

Parameters **worker_id** (*str*) – Worker id to be put on hold

initialize_scaling()
Compose the launch command and call the scale_out

This should be implemented in the child classes to take care of executor specific oddities.

outstanding

request_status_info()

scale_in(*blocks*)
Scale in the number of active blocks by specified amount.

The scale in method here is very rude. It doesn't give the workers the opportunity to finish current tasks or cleanup. This is tracked in issue #530

Raises NotImplementedError

scale_out(*blocks=1*)
Scales out the number of blocks by "blocks"

Raises NotImplementedError

scaling_enabled
Specify if scaling is enabled.

The callers of ParslExecutors need to differentiate between Executors and Executors wrapped in a resource provider

shutdown(*hub=True*, *targets='all'*, *block=False*)
Shutdown the executor, including all workers and controllers.

This is not implemented.

Kwargs:

- hub (Bool): Whether the hub should be shutdown, Default:True,
- targets (list of ints| 'all'): List of block id's to kill, Default:'all'
- block (Bool): To block for confirmations or not

Raises NotImplementedError

start()
Create the Interchange process and connect to it.

status()
Return status of all blocks.

submit(*bufs*, *task_id=None*)
Submits work to the the outgoing_q.

The outgoing_q is an external process listens on this queue for new work. This method behaves like a submit call as described here Python docs:

Parameters

- **– Pickled buffer with (b'<Function>', b'<args>', b'<kwargs>')** (*Bufs*) –
- **Returns** – Future

wait_for_endpoint()

weakref_cb(*q=None*)
We do not use this yet.

funcx.mock_broker package

Submodules

funcx.mock_broker.forwarder module

class funcx.mock_broker.forwarder.**Forwarder**(*task_q*, *result_q*, *executor*, *endpoint_id*, *logdir='forwarder'*, *logging_level=20*)
Bases: multiprocessing.context.Process

Forwards tasks/results between the executor and the queues

Tasks_Q Results_Q

^

|

V |

Executors

Todo : We need to clarify what constitutes a task that comes down the task pipe. Does it already have the code fragment? Or does that need to be sorted out from some DB ?

connection_info
Get the client ports to which the interchange must connect to

handle_app_update(*task_id*, *future*)
Triggered when the executor sees a task complete.

This can be further optimized at the executor level, where we trigger this or a similar function when we see a results item inbound from the interchange.

run()
Process entry point.

funcx.mock_broker.forwarder.**double**(*x*)

funcx.mock_broker.forwarder.**failer**(*x*)

funcx.mock_broker.forwarder.**spawn_forwarder**(*address*, *executor=None*, *task_q=None*, *result_q=None*, *endpoint_id=UUID('b69ddb5a-4c9e-42fa-8887-5ab463394c57')*, *logging_level=20*)

Spawns a forwarder and returns the forwarder process for tracking.

Parameters

- **address** (*str*) – IP Address to which the endpoint must connect
- **executor** (*Executor object. Optional*) – Executor object to be instantiated.
- **task_q** (*Queue object*) – Queue object matching funcx.queues.base.FuncxQueue interface
- **logging_level** (*int*) – Logging level as defined in the logging module. Default: logging.INFO (20)
- **endpoint_id** (*uuid string*) – Endpoint id for which the forwarder is being spawned.
- **Returns** – A Forwarder object

funcx.mock_broker.mock_broker module

The broker service

This REST service fields incoming registration requests from endpoints, creates an appropriate forwarder to which the endpoint can connect up.

funcx.mock_broker.mock_broker.**list_mappings**()

funcx.mock_broker.mock_broker.**register**()

Register an endpoint request

1. Start an executor client object corresponding to the endpoint
2. Pass connection info back as a json response.

funcx.mock_broker.mock_tester module

funcx.mock_broker.mock_tester.**test**(*address*)

funcx.mock_broker.test module

funcx.mock_broker.test.**double**(*x*)

funcx.mock_broker.test.**fail**(*x*)

funcx.mock_broker.test.**test_1**()

funcx.mock_broker.test.**test_2**()

funcx.mock_broker.test.**test_3**()

Module contents

funcx.providers package

Subpackages

funcx.providers.kubernetes package

Submodules

funcx.providers.kubernetes.kube module

class funcx.providers.kubernetes.kube.**KubernetesProvider**(*image: str*, *namespace: str = 'default'*, *nodes_per_block: int = 1*, *init_blocks: int = 4*, *min_blocks: int = 0*, *max_blocks: int = 10*, *max_cpu: float = 2*, *max_mem: str = '500Mi'*, *init_cpu: float = 1*, *init_mem: str = '250Mi'*, *parallelism: float = 1*, *worker_init: str = ''*, *pod_name: Optional[str] = None*, *user_id: Optional[str] = None*, *group_id: Optional[str] = None*, *run_as_non_root: bool = False*, *secret: Optional[str] = None*, *persistent_volumes: List[Tuple[str, str]] = []*)

Bases: parsl.providers.provider_base.ExecutionProvider, parsl.utils.RepresentationMixin

Kubernetes execution provider :param namespace: Kubernetes namespace to create deployments. :type namespace: str :param image: Docker image to use in the deployment. :type image: str :param nodes_per_block: Nodes to provision per block. :type nodes_per_block: int :param init_blocks: Number of blocks to provision at the start of the run. Default is 1. :type init_blocks: int :param min_blocks: Minimum number of blocks to maintain. :type min_blocks: int :param max_blocks: Maximum number of blocks to maintain. :type max_blocks: int :param max_cpu: CPU limits of the blocks (pods), in cpu units.

This is the cpu "limits" option for resource specification. Check kubernetes docs for more details. Default is 2.

Parameters

- **max_mem** (*str*) – Memory limits of the blocks (pods), in Mi or Gi. This is the memory "limits" option for resource specification on kubernetes. Check kubernetes docs for more details. Default is 500Mi.

- **init_cpu** (*float*) – CPU limits of the blocks (pods), in cpu units. This is the cpu "requests" option for resource specification. Check kubernetes docs for more details. Default is 1.
- **init_mem** (*str*) – Memory limits of the blocks (pods), in Mi or Gi. This is the memory "requests" option for resource specification on kubernetes. Check kubernetes docs for more details. Default is 250Mi.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. A parallelism value of 1 represents aggressive scaling where as many resources as possible are used; parallelism close to 0 represents the opposite situation in which as few resources as possible (i.e., min_blocks) are used.
- **worker_init** (*str*) – Command to be run first for the workers, such as *python start.py*.
- **secret** (*str*) – Docker secret to use to pull images
- **pod_name** (*str*) – The name for the pod, will be appended with a timestamp. Default is None, meaning parsl automatically names the pod.
- **user_id** (*str*) – Unix user id to run the container as.
- **group_id** (*str*) – Unix group id to run the container as.
- **run_as_non_root** (*bool*) – Run as non-root (True) or run as root (False).
- **persistent_volumes** (*list[(str, str)]*) – List of tuples describing persistent volumes to be mounted in the pod. The tuples consist of (PVC Name, Mount Directory).

cancel(*num_pods*, *task_type=None*)

Cancels the resources identified by the job_ids provided by the user.

Parameters **job_ids** (–) – A list of job identifiers

Returns

- A list of status from cancelling the job which can be True, False

Raises

- ExecutionProviderException or its subclasses

label

Provides the label for this provider

status(*job_ids*)

Get the status of a list of jobs identified by the job identifiers returned from the submit request. :param - job_ids: A list of job identifiers :type - job_ids: list

Returns

- A list of status from ['PENDING', 'RUNNING', 'CANCELLED', 'COMPLETED', 'FAILED', 'TIMEOUT'] corresponding to each job_id in the job_ids list.

Raises

- ExecutionProviderExceptions or its subclasses

submit(*cmd_string*, *tasks_per_node*, *task_type*, *job_name='FuncX'*)

Submit a job :param - cmd_string: (String) - Name of the container to initiate :param - tasks_per_node: command invocations to be launched per node :type - tasks_per_node: int

Kwargs:

- job_name (String): Name for job, must be unique

Returns At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

funcx.providers.kubernetes.template module

Module contents

Module contents

class funcx.providers.**KubernetesProvider**(*image: str*, *namespace: str = 'default'*, *nodes_per_block: int = 1*, *init_blocks: int = 4*, *min_blocks: int = 0*, *max_blocks: int = 10*, *max_cpu: float = 2*, *max_mem: str = '500Mi'*, *init_cpu: float = 1*, *init_mem: str = '250Mi'*, *parallelism: float = 1*, *worker_init: str = ''*, *pod_name: Optional[str] = None*, *user_id: Optional[str] = None*, *group_id: Optional[str] = None*, *run_as_non_root: bool = False*, *secret: Optional[str] = None*, *persistent_volumes: List[Tuple[str, str]] = []*)

Bases: parsl.providers.provider_base.ExecutionProvider, parsl.utils.RepresentationMixin

Kubernetes execution provider :param namespace: Kubernetes namespace to create deployments. :type namespace: str :param image: Docker image to use in the deployment. :type image: str :param nodes_per_block: Nodes to provision per block. :type nodes_per_block: int :param init_blocks: Number of blocks to provision at the start of the run. Default is 1. :type init_blocks: int :param min_blocks: Minimum number of blocks to maintain. :type min_blocks: int :param max_blocks: Maximum number of blocks to maintain. :type max_blocks: int :param max_cpu: CPU limits of the blocks (pods), in cpu units.

This is the cpu "limits" option for resource specification. Check kubernetes docs for more details. Default is 2.

Parameters

- **max_mem** (*str*) – Memory limits of the blocks (pods), in Mi or Gi. This is the memory "limits" option for resource specification on kubernetes. Check kubernetes docs for more details. Default is 500Mi.
- **init_cpu** (*float*) – CPU limits of the blocks (pods), in cpu units. This is the cpu "requests" option for resource specification. Check kubernetes docs for more details. Default is 1.
- **init_mem** (*str*) – Memory limits of the blocks (pods), in Mi or Gi. This is the memory "requests" option for resource specification on kubernetes. Check kubernetes docs for more details. Default is 250Mi.
- **parallelism** (*float*) – Ratio of provisioned task slots to active tasks. A parallelism value of 1 represents aggressive scaling where as many resources as possible are used; parallelism close to 0 represents the opposite situation in which as few resources as possible (i.e., min_blocks) are used.
- **worker_init** (*str*) – Command to be run first for the workers, such as *python start.py*.

- **secret** (*str*) – Docker secret to use to pull images
- **pod_name** (*str*) – The name for the pod, will be appended with a timestamp. Default is None, meaning parsl automatically names the pod.
- **user_id** (*str*) – Unix user id to run the container as.
- **group_id** (*str*) – Unix group id to run the container as.
- **run_as_non_root** (*bool*) – Run as non-root (True) or run as root (False).
- **persistent_volumes** (*list[(str, str)]*) – List of tuples describing persistent volumes to be mounted in the pod. The tuples consist of (PVC Name, Mount Directory).

cancel(*num_pods*, *task_type=None*)
Cancels the resources identified by the job_ids provided by the user.

Parameters **job_ids** (–) – A list of job identifiers

Returns

- A list of status from cancelling the job which can be True, False

Raises

- ExecutionProviderException or its subclasses

label
Provides the label for this provider

status(*job_ids*)
Get the status of a list of jobs identified by the job identifiers returned from the submit request. :param - job_ids: A list of job identifiers :type - job_ids: list

Returns

- A list of status from ['PENDING', 'RUNNING', 'CANCELLED', 'COMPLETED', 'FAILED', 'TIMEOUT'] corresponding to each job_id in the job_ids list.

Raises

- ExecutionProviderExceptions or its subclasses

submit(*cmd_string*, *tasks_per_node*, *task_type*, *job_name='FuncX'*)
Submit a job :param - cmd_string: (String) - Name of the container to initiate :param - tasks_per_node: command invocations to be launched per node :type - tasks_per_node: int

Kwargs:

- job_name (String): Name for job, must be unique

Returns At capacity, cannot provision more - job_id: (string) Identifier for the job

Return type

- None

funcx.queues package

Subpackages

funcx.queues.redis package

Submodules

funcx.queues.redis.redis_q module

class funcx.queues.redis.redis_q.**RedisQueue**(*prefix*, *hostname*, *port=6379*)
Bases: *funcx.queues.base.FuncxQueue*

A basic redis queue

The queue only connects when the *connect* method is called to avoid issues with passing an object across processes.

Parameters

- **hostname** (*str*) – Hostname of the redis server
- **port** (*int*) – Port at which the redis server can be reached. Default: 6379

connect()
Connects to the Redis server

get(*timeout=1*)
Get an item from the redis queue

Parameters **timeout** (*int*) – Timeout for the blocking get in seconds

is_connected
Returns the connected status of the queue.

Returns

Return type Bool

put(*key*, *payload*)
Put's the key:payload into a dict and pushes the key onto a queue :param key: The task_id to be pushed :type key: str :param payload: Dict of task information to be stored :type payload: dict

funcx.queues.redis.redis_q.**test**()

Module contents

Submodules

funcx.queues.base module

class funcx.queues.base.**FuncxQueue**
Bases: object

Queue interface required by the Forwarder

This is a metaclass that only enforces concrete implementations of functionality by the child classes.

connect(**args*, ***kwargs*)
Connects and creates the queue. The queue is not active until this is called

get(**args*, ***kwargs*)
Get an item from the Queue

is_connected
Returns the connected status of the queue.

Returns

Return type Bool

put(**args*, ***kwargs*)
Put an item into the Queue

exception funcx.queues.base.**NotConnected**(*queue*)
Bases: *funcx.errors.FuncxError*

Queue is not connected/active

Module contents

class funcx.queues.**RedisQueue**(*prefix*, *hostname*, *port=6379*)
Bases: *funcx.queues.base.FuncxQueue*

A basic redis queue

The queue only connects when the *connect* method is called to avoid issues with passing an object across processes.

Parameters

- **hostname** (*str*) – Hostname of the redis server
- **port** (*int*) – Port at which the redis server can be reached. Default: 6379

connect()
Connects to the Redis server

get(*timeout=1*)
Get an item from the redis queue

Parameters **timeout** (*int*) – Timeout for the blocking get in seconds

is_connected
Returns the connected status of the queue.

Returns

Return type Bool

put(*key*, *payload*)
Put's the key:payload into a dict and pushes the key onto a queue :param key: The task_id to be pushed :type key: str :param payload: Dict of task information to be stored :type payload: dict

funcx.sdk package

Subpackages

funcx.sdk.utils package

Submodules

funcx.sdk.utils.futures module

Tools for dealing with asynchronous execution

Credit: Logan Ward

class funcx.sdk.utils.futures.**FuncXFuture**(*client*, *task_id: str*, *ping_interval: float*)
Bases: concurrent.futures._base.Future

Utility class for simplifying asynchronous execution in funcX

cancel()
Stop the execution of the function

running()
Return True if the future is currently executing.

funcx.sdk.utils.throttling module

exception funcx.sdk.utils.throttling.**MaxRequestSizeExceeded**
Bases: *funcx.sdk.utils.throttling.ThrottlingException*

exception funcx.sdk.utils.throttling.**MaxRequestsExceeded**
Bases: *funcx.sdk.utils.throttling.ThrottlingException*

class funcx.sdk.utils.throttling.**ThrottledBaseClient**(**args*, ***kwargs*)
Bases: globus_sdk.base.BaseClient

Throttled base client allows for two different types of Throttling. Note, this is 'polite' client side throttling, to avoid well intentioned users from mistakenly harming the funcX service, this does not prevent DOS attacks.

Request Flood Throttling: Restricts the number of requests that can be made in a given time period, and raises an exception when that has been exceeded

Request Size Throttling: Restricts the size of the payload that can be sent to the FuncX web service.

Both of these raise exceptions when the values have been exceeded. Both can be caught with ThrottlingException. Throttling can also be turned off with:

cli.throttling_enabled = False

DEFAULT_MAX_REQUESTS = 5

DEFAULT_MAX_REQUEST_SIZE = 2097152

throttle_max_requests()

throttle_request_size(**request_args*, ***request_kwargs*)

exception funcx.sdk.utils.throttling.**ThrottlingException**
Bases: Exception

Module contents

Submodules

funcx.sdk.client module

class funcx.sdk.client.**FuncXClient**(*http_timeout=None*, *funcx_home='~/.funcx'*, *force_login=False*, *fx_authorizer=None*, *funcx_service_address='https://dev.funcx.org/api/v1'*, ***kwargs*)
Bases: *funcx.sdk.utils.throttling.ThrottledBaseClient*

Main class for interacting with the funcX service

Holds helper operations for performing common tasks with the funcX service.

CLIENT_ID = '4cf29807-cf21-49ec-9443-ff9a3fb9f81c'

TOKEN_DIR = '/home/docs/.funcx/credentials'

TOKEN_FILENAME = 'funcx_sdk_tokens.json'

add_to_whitelist(*endpoint_id*, *function_ids*)
Adds the function to the endpoint's whitelist

Parameters

- **endpoint_id** (*str*) – The uuid of the endpoint
- **function_ids** (*list*) – A list of function id's to be whitelisted

Returns The response of the request

Return type json

batch_run(*batch*)
Initiate a batch of tasks to funcX

Parameters **batch** (*a Batch object*) –

Returns **task_ids**

Return type a list of UUID strings that identify the tasks

create_batch()
Create a Batch instance to handle batch submission in funcX

Returns Status block containing "status" key.

Return type Batch instance

delete_from_whitelist(*endpoint_id*, *function_ids*)
List the endpoint's whitelist

Parameters

- **endpoint_id** (*str*) – The uuid of the endpoint
- **function_ids** (*list*) – A list of function id's to be whitelisted

Returns The response of the request

Return type json

get_batch_result(*task_id_list*)
Request results for a batch of task_ids

get_batch_status(*task_id_list*)
Request status for a batch of task_ids

get_container(*container_uuid*, *container_type*)
Get the details of a container for staging it locally.

Parameters

- **container_uuid** (*str*) – UUID of the container in question
- **container_type** (*str*) – The type of containers that will be used (Singularity, Shifter, Docker)

Returns The details of the containers to deploy

Return type dict

get_containers(*name*, *description=None*)
Register a DLHub endpoint with the funcX service and get the containers to launch.

Parameters

- **name** (*str*) – Name of the endpoint
- **description** (*str*) – Description of the endpoint

Returns The port to connect to and a list of containers

Return type int

get_endpoint_status(*endpoint_uuid*)
Get the status reports for an endpoint.

Parameters **endpoint_uuid** (*str*) – UUID of the endpoint in question

Returns The details of the endpoint's stats

Return type dict

get_result(*task_id*)
Get the result of a funcX task

Parameters **task_id** (*str*) – UUID of the task

Returns **Result obj**

Return type If task completed

Raises Exception obj: Exception due to which the task failed

get_task_status(*task_id*)
Get the status of a funcX task.

Parameters **task_id** (*str*) – UUID of the task

Returns Status block containing "status" key.

Return type dict

get_whitelist(*endpoint_id*)
List the endpoint's whitelist

Parameters **endpoint_id** (*str*) – The uuid of the endpoint

Returns The response of the request

Return type json

logout()
Remove credentials from your local system

map_run(**args*, *endpoint_id=None*, *function_id=None*, *asynchronous=False*, ***kwargs*)
Initiate an invocation

Parameters

- ***args** (*Any*) – Args as specified by the function signature
- **endpoint_id** (*uuid str*) – Endpoint UUID string. Required
- **function_id** (*uuid str*) – Function UUID string. Required
- **asynchronous** (*bool*) – Whether or not to run the function asynchronously

Returns

- **task_id** (*str*)
- *UUID string that identifies the task*

register_container(*location*, *container_type*, *name=''*, *description=''*)
Register a container with the funcX service.

Parameters

- **location** (*str*) – The location of the container (e.g., its docker url). Required
- **container_type** (*str*) – The type of containers that will be used (Singularity, Shifter, Docker). Required
- **name** (*str*) – A name for the container. Default = ''
- **description** (*str*) – A description to associate with the container. Default = ''

Returns The id of the container

Return type str

register_endpoint(*name*, *endpoint_uuid*, *description=None*)
Register an endpoint with the funcX service.

Parameters

- **name** (*str*) – Name of the endpoint
- **endpoint_uuid** (*str*) – The uuid of the endpoint
- **description** (*str*) – Description of the endpoint

Returns

{'endopoint_id' [<>,] 'address' : <>, 'client_ports': <>}

Return type A dict

register_function(*function*, *function_name=None*, *container_uuid=None*, *description=None*, *public=False*, *group=None*)
Register a function code with the funcX service.

Parameters

- **function** (*Python Function*) – The function to be registered for remote execution
- **function_name** (*str*) – The entry point (function name) of the function. Default: None
- **container_uuid** (*str*) – Container UUID from registration with funcX
- **description** (*str*) – Description of the file
- **public** (*bool*) – Whether or not the function is publicly accessible. Default = False
- **group** (*str*) – A globus group uuid to share this function with

Returns **function uuid** – UUID identifier for the registered function

Return type str

run(**args*, *endpoint_id=None*, *function_id=None*, ***kwargs*)
Initiate an invocation

Parameters

- ***args** (*Any*) – Args as specified by the function signature

- **endpoint_id** (*uuid str*) – Endpoint UUID string. Required
- **function_id** (*uuid str*) – Function UUID string. Required
- **asynchronous** (*bool*) – Whether or not to run the function asynchronously

Returns

- **task_id** (*str*)
- *UUID string that identifies the task*

update_table(*return_msg*, *task_id*)
Parses the return message from the service and updates the internal func_tables

Parameters

- **return_msg** (*str*) – Return message received from the funcx service
- **task_id** (*str*) – task id string

funcx.sdk.config module

funcx.sdk.config.**write_option**(*option*, *value*)
Write an option to disk – doesn't handle config reloading

funcx.sdk.config.**lookup_option**(*option*)

funcx.sdk.config.**remove_option**(*option*)

funcx.sdk.config.**internal_auth_client**()
Get the globus native app client.

Returns

funcx.sdk.config.**check_logged_in**()
Check if the user is already logged in.

Returns

funcx.sdk.config.**safeprint**(*s*)
Catch print errors.

Parameters **s** –

Returns

funcx.sdk.config.**format_output**(*dataobject*)
Use safe print to make sure jobs are correctly printed.

Parameters **dataobject** –

Returns

funcx.sdk.version module

Module contents

funcx.serialize package

Submodules

funcx.serialize.base module

exception funcx.serialize.base.**DeserializationError**(*reason*)
Bases: Exception

Base class for all deserialization errors

class funcx.serialize.base.**RemoteExceptionWrapper**(*e_type*, *e_value*, *traceback*)
Bases: object

reraise()

class funcx.serialize.base.**SerializerError**(*reason*)
Bases: object

class funcx.serialize.base.**fxPicker_enforcer**
Bases: object

Ensure that any concrete class will have the serialize and deserialize methods

deserialize(*payload*)

serialize(*data*)

class funcx.serialize.base.**fxPicker_shared**
Bases: object

Adds shared functionality for all serializer implementations

check(*payload*)

chomp(*payload*)
If the payload starts with the identifier, return the remaining block

Parameters **payload** (*str*) – Payload blob

identifier
Get the identifier of the serialization method

Returns **identifier**

Return type str

funcx.serialize.concretes module

funcx.serialize.concretes.**bar**(*x*, *y={'a': 3}*)

class funcx.serialize.concretes.**code_pickle**
Bases: *funcx.serialize.base.fxPicker_shared*

deserialize(*payload*)

serialize(*data*)

class funcx.serialize.concretes.**code_text_dill**
Bases: *funcx.serialize.base.fxPicker_shared*

We use dill to get the source code out of the function object and then exec the function body to load it in. The function object is then returned by name.

deserialize(*payload*)

serialize(*data*)

class funcx.serialize.concretes.**code_text_inspect**
Bases: *funcx.serialize.base.fxPicker_shared*

We use dill to get the source code out of the function object and then exec the function body to load it in. The function object is then returned by name.

deserialize(*payload*)

serialize(*data*)

class funcx.serialize.concretes.**json_base64**
Bases: *funcx.serialize.base.fxPicker_shared*

deserialize(*payload*)

serialize(*data*)

class funcx.serialize.concretes.**pickle_base64**
Bases: *funcx.serialize.base.fxPicker_shared*

deserialize(*payload*)

serialize(*data*)

funcx.serialize.facade module

class funcx.serialize.facade.**FuncXSerializer**
Bases: object

Information that we want to be able to ship around:

- Function run information ->
- Function id <—> Container id ? (is there a 1-1 mapping here?)
- Container id
- Endpoint id
- Function body
- Potentially in a byte compiled form ?
- All parameters ->
- Args + Kwargs

From the client side. At function invocation we need to capture

deserialize(*payload*)

Parameters **payload** (*str*) – Payload object to be deserialized

pack_buffers(*buffers*)

buffers : list of

terminated strings

serialize(*data*)

unpack_and_deserialize(*packed_buffer*)
Unpacks a packed buffer and returns the deserialized contents :param packed_buffers: :type packed_buffers: packed buffer as string

unpack_buffers(*packed_buffer*)

Parameters **packed_buffers** (*packed buffer as string*) –

Module contents

funcx.strategies package

Submodules

funcx.strategies.base module

class funcx.strategies.base.**BaseStrategy**(**args*, *threshold=20*, *interval=5*)
Bases: object

Implements threshold-interval based flow control.

The overall goal is to trap the flow of apps from the workflow, measure it and redirect it the appropriate executors for processing.

This is based on the following logic:

```
BEGIN (INTERVAL, THRESHOLD, callback) :
    start = current_time()

    while (current_time()-start < INTERVAL) :
         count = get_events_since(start)
         if count >= THRESHOLD :
             break

    callback()
```

This logic ensures that the callbacks are activated with a maximum delay of *interval* for systems with infrequent events as well as systems which would generate large bursts of events.

Once a callback is triggered, the callback generally runs a strategy method on the sites available as well asqeuque

TODO: When the debug logs are enabled this module emits duplicate messages. This issue needs more debugging. What I've learnt so far is that the duplicate messages are present only when the timer thread is started, so this could be from a duplicate logger being added by the thread.

close()
Merge the threads and terminate.

make_callback(*kind=None*)
Makes the callback and resets the timer.

KWargs:

- kind (str): Default=None, used to pass information on what triggered the callback

notify(*event_id*)
Let the FlowControl system know that there is an event.

This method is to be called from the Interchange to notify the flowcontrol

start(*interchange*)
Actually start the strategy :param interchange: Interchange to bind the strategy to :type interchange: funcx.executors.high_throughput.interchange.Interchange

strategize(**args*, ***kwargs*)
Strategize is called everytime the threshold or the interval is hit

class funcx.strategies.base.**Timer**(*callback*, **args*, *interval=5*)
Bases: object

This timer is a simplified version of the FlowControl timer. This timer does not employ notify events.

This is based on the following logic :

```
BEGIN (INTERVAL, THRESHOLD, callback) :
    start = current_time()

    while (current_time()-start < INTERVAL) :
         wait()
         break

    callback()
```

close()
Merge the threads and terminate.

make_callback(*kind=None*)
Makes the callback and resets the timer.

funcx.strategies.kube_simple module

class funcx.strategies.kube_simple.**KubeSimpleStrategy**(**args*, *threshold=20*, *interval=1*, *max_idletime=60*)
Bases: *funcx.strategies.base.BaseStrategy*

Implements the simple strategy for Kubernetes

strategize(**args*, ***kwargs*)
Strategize is called everytime the threshold or the interval is hit

funcx.strategies.simple module

class funcx.strategies.simple.**SimpleStrategy**(**args*, *threshold=20*, *interval=1*, *max_idletime=60*)
Bases: *funcx.strategies.base.BaseStrategy*

Implements the simple strategy

strategize(**args*, ***kwargs*)
Strategize is called everytime the threshold or the interval is hit

funcx.strategies.test module

Module contents

class funcx.strategies.**BaseStrategy**(**args*, *threshold=20*, *interval=5*)
Bases: object

Implements threshold-interval based flow control.

The overall goal is to trap the flow of apps from the workflow, measure it and redirect it the appropriate executors for processing.

This is based on the following logic:

```
BEGIN (INTERVAL, THRESHOLD, callback) :
    start = current_time()

    while (current_time()-start < INTERVAL) :
         count = get_events_since(start)
         if count >= THRESHOLD :
             break

    callback()
```

This logic ensures that the callbacks are activated with a maximum delay of *interval* for systems with infrequent events as well as systems which would generate large bursts of events.

Once a callback is triggered, the callback generally runs a strategy method on the sites available as well asqeuque

TODO: When the debug logs are enabled this module emits duplicate messages. This issue needs more debugging. What I've learnt so far is that the duplicate messages are present only when the timer thread is started, so this could be from a duplicate logger being added by the thread.

close()
Merge the threads and terminate.

make_callback(*kind=None*)
Makes the callback and resets the timer.

KWargs:

- kind (str): Default=None, used to pass information on what triggered the callback

notify(*event_id*)
Let the FlowControl system know that there is an event.

This method is to be called from the Interchange to notify the flowcontrol

start(*interchange*)
Actually start the strategy :param interchange: Interchange to bind the strategy to :type interchange: funcx.executors.high_throughput.interchange.Interchange

strategize(**args*, ***kwargs*)
Strategize is called everytime the threshold or the interval is hit

class funcx.strategies.**SimpleStrategy**(**args*, *threshold=20*, *interval=1*, *max_idletime=60*)
Bases: *funcx.strategies.base.BaseStrategy*

Implements the simple strategy

strategize(**args*, ***kwargs*)
Strategize is called everytime the threshold or the interval is hit

class funcx.strategies.**KubeSimpleStrategy**(**args*, *threshold=20*, *interval=1*, *max_idletime=60*)
Bases: *funcx.strategies.base.BaseStrategy*

Implements the simple strategy for Kubernetes

strategize(**args*, ***kwargs*)
Strategize is called everytime the threshold or the interval is hit

5.1.2 Submodules

5.1.3 funcx.config module

class funcx.config.**Config**(*provider=LocalProvider(channel=LocalChannel(envs={}, script_dir=None, userhome='/home/docs/checkouts/readthedocs.org/user_builds/funcx/checkou), cmd_timeout=30, init_blocks=4, launcher=SingleNodeLauncher(), max_blocks=10, min_blocks=0, move_files=None, nodes_per_block=1, parallelism=1, walltime='00:15:00', worker_init="), scaling_enabled=True, worker_ports=None, worker_port_range=(54000, 55000), strategy=<funcx.strategies.simple.SimpleStrategy object>, max_workers_per_node=inf, cores_per_worker=1.0, mem_per_worker=None, launch_cmd=None, worker_mode='no_container', scheduler_mode='hard', container_type=None, prefetch_capacity=10, heartbeat_period=2, heartbeat_threshold=10, poll_period=10, working_dir=None, worker_debug=False*)

Bases: parsl.utils.RepresentationMixin

Specification of FuncX configuration options.

Parameters

- **max_workers_per_node** (*int*) – Maximum # of worker per node. Default: inf
- **cores_per_worker** (*float*) – cores to be assigned to each worker. Oversubscription is possible by setting cores_per_worker < 1.0. Default=1
- **mem_per_worker** (*float*) – GB of memory required per worker. If this option is specified, the node manager will check the available memory at startup and limit the number of workers such that the there's sufficient memory for each worker. Default: None
- **working_dir** (*str*) – Working dir to be used by the executor. Default to the endpoint directory if not specified
- **worker_debug** (*Bool*) – Enables worker debug logging.
- **worker_mode** (*str*) – Select the mode of operation from no_container, singularity_reuse, singularity_single_use Default: no_container
- **scheduler_mode** (*str*) – Select the mode of how the container is managed from hard, soft Default: hard
- **container_type** (*str*) – Select the type of container from Docker, Singularity, Shifter Default: None
- **scaling_enabled** (*Bool*) – Allow Interchange to manage resource provisioning. If set to False, interchange will not do any scaling. Default: True

5.1.4 funcx.errors module

exception funcx.errors.**FuncXUnreachable**(*address*)

Bases: *funcx.errors.FuncxError*

FuncX remote service is unreachable

exception funcx.errors.**FuncxError**

Bases: Exception

Base class for all funcx exceptions

exception funcx.errors.**MalformedResponse**(*response*)
Bases: *funcx.errors.FuncxError*

FuncX remote service responded with a Malformed Response

exception funcx.errors.**RegistrationError**(*reason*)
Bases: *funcx.errors.FuncxError*

Registering the endpoint has failed

5.1.5 funcx.version module

Set module version.

<Major>.<Minor>.<maintenance>[alpha/beta/..] Alphas will be numbered like this -> 0.4.0a0

5.1.6 Module contents

funcX : Fast function serving for clouds, clusters and supercomputers.

funcx.**set_file_logger**(*filename*, *name='funcx'*, *level=10*, *format_string=None*)
Add a stream log handler.

Parameters

- **filename** (–) – Name of the file to write logs to
- **name** (–) – Logger name
- **level** (–) – Set the logging level.
- **format_string** (–) – Set the format string

Returns

- None

funcx.**set_stream_logger**(*name='funcx'*, *level=10*, *format_string=None*)
Add a stream log handler.

Parameters

- **name** (–) – Set the logger name.
- **level** (–) – Set to logging.DEBUG by default.
- **format_string** (–) – Set to None by default.

Returns

- None

CHAPTER 6

Debugging

To sidestep the FuncX web-service the endpoint can be forced to connect directly to a debug-forwarder. To enable this behavior update *~/.funcx/config*

```
* 'broker_address' : Address at which the broker is listening, eg: "http://127.0.0.1:
↪50005"
* 'broker_test' : True
* 'redis_host' : Point redis host to a locally running redis server. Eg: "127.0.0.1"
```

6.1 Setting up the forwarder locally

You can run the forwareder-service in debug mode on your local system and skip the web-service entirely. For this, make sure you have the redis package installed and running. You can check this by running:

```
>>> redis-cli
```

This should output a prompt that says : 127.0.0.1:6379. This string needs to match.

Now, you can start the forwarder service for testing:

```
>>> forwarder-service --address 127.0.0.1 --port 50005 --debug
```

Once you have this running, we can update the endpoint configs to point to this local service.

CHAPTER 7

Indices and tables

- genindex
- search

Python Module Index

f

Index

D

E

F

G

H

I

J

K

L

M

N