Dedalus Documentation

Release 0.1.4

Esmerald Aliaj, Georgia Dimaki

October 29, 2018

Table of Contents

Contents:

CHAPTER 1

Broker

class dedalus.mnbroker.**MNBroker** (*context*, *main_ep*, *opt_ep=None*, *service_q=None*, *data_q=None*)

The Dedalus broker class.

The broker routes messages from clients to appropriate nodes/workers based on the requested data. It will also allow workers to register to different services as well as files.

This base class defines the overall functionality and the API. Subclasses are meant to implement additional features.

Note The workers will *always* be served by the *main_ep* endpoint.

In a two-endpoint setup clients will be handled via the *opt_ep* endpoint.

Parameters

- **context** (*zmq.Context*) -- the context to use for socket creation.
- **main_ep** (*str*) -- the primary endpoint for workers and clients.
- **opt_ep** (*str*) -- is an optional 2nd endpoint.
- **service_q** (*class*) -- the class to be used for the service worker-queue.
- **data_q** -- the class to be used for the data-queue.

change_worker_status (*wid*, *status*)

Change the status of the worker with the given id.

Parameters

- **wid** (*str*) -- worker id
- **status** (*bytearray*) -- new worker status

Return type None

client_response (*rp*, *service*, *cmd*, *wid*, *msg*)

Package and send reply to client.

The message will contain the protocol used to serve this update, the service used, as well as echo back the requested command id and worker id.

Parameters

- **rp** (*list of str*) -- return address stack
- **service** (*str*) -- name of service
- **cmd** (*str*) -- id of the operation requested by the client
- **wid** (*str*) -- id of the worker that is replying
- **msg** (*list of str*) -- message parts

Return type None

collect_workers_info ( )
Send a special request to all known workers to trigger an update event.

Return type None

disconnect (*wid*)
Send disconnect command and unregister worker.
If the worker id is not registered, nothing happens.

Parameters **wid** (*str*) -- the worker id.

Return type None

find_worker (*wid*, *service*)
Find a worker with the given id.

Parameters
- **wid** (*str*) -- data id
- **service** (*str*) -- service the worker supports

Return type str

get_workers_info ( )
Return a list with the information of all current workers.

Return type dict

on_client (*proto*, *rp*, *msg*)
Method called on client message.
Frame 0 of msg is the requested service. The remaining frames are the request to forward to the worker.

Note If the service is unknown to the broker the message is ignored.

Note If currently no worker is available for a known service, the message is queued for later delivery.

If a worker is available for the requested service, the message is repackaged and sent to the worker. The worker in question is removed from the pool of available workers.
If the service name starts with '-ho., the message is passed to the internal HO_ handler.

Parameters
- **proto** (*str*) -- the protocol id sent
- **rp** (*list of str*) -- return address stack
- **msg** (*list of str*) -- message parts

Return type None

on_disconnect (*rp*, *msg*)
Process worker DISCONNECT command.
Unregisters the worker who sent this message.

Parameters
- **rp** (*list of str*) -- return address stack
- **msg** (*list of str*) -- message parts

Return type None

on_heartbeat (*rp*, *msg*)
Process worker HEARTBEAT command.

Parameters
- **rp** (*list of str*) -- return address stack
- **msg** (*list of str*) -- message parts

Return type None

on_ho (*rp*, *service*, *msg*)
Process HO request.
For now only ho.service and ho.stat are handled. Offering statistics on all connected workers.

Note All operations that involve the broker only and need to be exposed through the API, should be added here.

Parameters
- **rp** (*list of str*) -- return address stack
- **service** (*str*) -- the protocol id sent
- **msg** (*list of str*) -- message parts

Return type None

on_message (*msg*)
Processes given message.
Decides what kind of message it is -- client or worker -- and calls the appropriate method. If unknown, the message is ignored.

Parameters **msg** (*list of str*) -- message parts

Return type None

on_ready (*rp*, *msg*)
Process worker READY command.
Registers the worker for a service.

Parameters
- **rp** (*list of str*) -- return address stack
- **msg** (*list of str*) -- message parts

Return type None

on_reply (*rp*, *msg*)
Process worker REPLY command.
Route the *msg* to the client given by the address(es) in front of *msg*.

Parameters
- **rp** (*list of str*) -- return address stack
- **msg** (*list of str*) -- message parts

Return type None

on_timer ( )
Method called on timer expiry.
Checks which workers are dead and unregisters them.

Return type None

on_worker (*proto*, *rp*, *msg*)
Method called on worker message.
Frame 0 of msg is the command id. The remaining frames depend on the command.
This method determines the command sent by the worker and calls the appropriate method. If the command is unknown the message is ignored and a DISCONNECT is sent.

Parameters
- **proto** (*str*) -- the protocol id sent
- **rp** (*list of str*) -- return address stack
- **msg** (*list of str*) -- message parts

Return type None

register_worker (*wid*, *service*, *worker_type*, *address*, *protocols*)
Register the worker id and add it to the given service.
Does nothing if worker is already known.

Parameters
- **wid** (*str*) -- the worker id.
- **service** (*str*) -- the service name.
- **worker_type** (*str*) -- the type of the worker.
- **address** (*str*) -- the ipv4 or upv6 address of the worker.
- **protocols** (*str*) -- the routing protocols reported by the worker.

Return type None

register_worker_info (*wid*, *service=b'broker'*, *status=b'online'*, *worker_type=b'00'*, *address='10.10.0.197'*, *protocols=None*)
Update the worker info list.

Parameters
- **wid** (*str*) -- the worker id.
- **service** (*str*) -- the service name.
- **status** (*str*) -- the current network status of the worker.
- **worker_type** (*str*) -- the specific worker type.
- **address** (*str*) -- the ipv4 or ipv6 of this worker.
- **protocols** (*str*) -- the routing protocols reported by the worker.

Return type None

reset_node_info (*wid*, *status=b'lost'*)
Clear the Brokers' internal worker list.

Parameters
- **wid** (*str*) -- the worker id.
- **status** (*bytearray*) -- worker status from the available types in config.py

Returns None

send_to_all_workers (*rp*, *msg_type=b'wr01'*)
Send a message to all workers.

Parameters
- **rp** (*list of bytearray*) -- a list with the addresses of the intended recipients
- **msg_type** (*bytearray*) -- command to be executed by the workers

Returns None

shutdown ( )
Shutdown broker.
Will unregister all workers, stop all timers and ignore all further messages.

> **Warning** The instance MUST not be used after *shutdown()* has been called.

Return type None

unregister_worker (*wid*)
Unregister the worker with the given id and stop all timers for the worker.
If the worker id is not registered, nothing happens.

Parameters **wid** (*str*) -- the worker id.

Return type None

update_worker_info (*wid*, *data*)
Update the internal worker data for the worker with id *wid* to include information given in *data*.

Parameters
- **wid** (*str*) -- worker id
- **data** (*dict*) -- arbitrary information we want to associate with this node

Return type None

class dedalus.mnbroker.**ServiceQueue**
Class defining the Queue interface for workers for a service.
The methods on this class are the only ones used by the broker.

get ( )
Get the next worker from the queue.

Return type *WorkerTracker*

put (*wid*)
Put a worker in the queue.
Nothing will happen if the worker is already in queue.

Parameters **wid** (*str*) -- the workers id

remove (*wid*)
Remove a worker from the queue.

Parameters **wid** (*str*) -- the workers id

class dedalus.mnbroker.**WorkerTracker** (*proto*, *wid*, *service*, *stream*)
Helper class to represent a worker in the broker.
Instances of this class are used to track the state of the attached worker and carry the timers for incoming and outgoing heartbeats.

Parameters
- **proto** (*str*) -- the worker protocol id.
- **wid** (*str*) -- the worker id.
- **service** (*str*) -- service this worker serves
- **stream** (*ZMQStream*) -- the ZMQStream used to send messages

is_alive ( )
Returns True when the worker is considered alive.

Returns bool

on_heartbeat ( )
Called when a heartbeat message from the worker was received.
Sets current retries to HB_RETRIES.

send_hb ( )
Called on every HB_INTERVAL.
Decrements the current retries count by one.
Sends heartbeat to worker.

set_stream (*stream*)
Assign any of the available Dedalus streams to this worker.

Parameters **stream** (*ZMQStream*) -- A stream identified by a port and IP address

Returns None

shutdown ( )
Cleanup worker.
Stops timer.

Returns None

1.1 Dedalus broker

class dedalus.mnbroker_runner.**DedalusRunner** (*context*, *main_ep*, *opt_ep=None*, *service_q=None*, *data_q=None*)
A class that inherits *MNBroker*.
All additional functionality that is specific to Dedalus is added here.

dedalus.mnbroker_runner.**run** (*address*, *optional_address=None*)
Run an instance of a Dedalus broker.

Parameters
- **address** (*str*) -- the address to be used for the main stream
- **optional_address** (*str*) -- the address to be used for the optional client stream

Returns None

CHAPTER 2

Worker

exception dedalus.mnworker.**ConnectionNotReadyError**
Exception raised when attempting to use the MNWorker before the handshake took place.

class dedalus.mnworker.**DataTracker** (*hid*, *service*, *contents*)
Helper class to represent a cached data object in the worker.

Parameters
- **hid** (*str*) -- the worker id.
- **service** (*str*) -- service this data belongs to
- **contents** (*bytearray*) -- contents of the hashed data

get_size ( )
Will return the size in bytes of the data held by this object.

Return type int

class dedalus.mnworker.**MNWorker** (*context*, *endpoint*, *service*, *worker_type*, *address*, *protocols*)
Class for the MN worker side.
Provides a send method with optional timeout parameter.
Will use a timeout to indicate a broker failure.

Parameters
- **context** (*zmq.Context*) -- the context to use for socket creation.
- **endpoint** (*str*) -- endpoint to connect to.
- **service** (*byte-string*) -- the name of the service we support.

on_request (*msg*)
Public method called when a request arrived.

Parameters **msg** (*a list of byte-strings*) -- a list w/ the message parts
Must be overloaded to provide support for various services!

reply (*msg*)
Send the given message.

Parameters **msg** (*can either be a byte-string or a list of byte-strings*) -- full message to send.

Raises ConnectionNotReadyError

send_hb ( )
Construct and send HB message to broker.

shutdown ( )
Method to deactivate the worker connection completely.
Will delete the stream and the underlying socket.

exception dedalus.mnworker.**MissingHeartbeat**
Exception raised when a heartbeat was not received on time.

2.1 Dedalus worker

class dedalus.mnworker_runner.**WorkerRunner** (*context*, *endpoint*, *service*)
Inherits *MNWorker* and provides Dedalus specific functionality to the worker instance.

Parameters
- **context** (*Context*) -- context under which the worker will operate, providing the operation loop
- **endpoint** (*str*) -- the endpoint of the broker we want to connect to
- **service** (*list of str*) -- the list of services this worker will suport

dump (*msg*)
Return system info, including routing tables, hardware information as well as any other information provided by the underlying protocol modules.

Return type dict

local_commands (*cmd*, *msg*)
Process a request for a local function.

Parameters
- **cmd** (*bytearray*) -- opcode for the command to execute
- **msg** (*list of str*) -- additional arguments of the request

Return type None

on_request (*msg*)
Process a request command.

Parameters **msg** (*list of str*) -- contents of the request

Returns None

shutdown ( )
Close this worker after also shutting down all currently open protocol modules.

Returns None

dedalus.mnworker_runner.**run** (*address*)
Run an instance of a Dedalus worker.

Parameters **address** (*str*) -- the address of the broker to connect to

Returns None

CHAPTER 3

Client

exception dedalus.mnclient.**InvalidStateError**
Exception raised when the requested action is not available due to socket state.

class dedalus.mnclient.**MNClient** (*context*, *endpoint*, *service*)
Class for the MN client side.
Thin asynchronous encapsulation of a zmq.REQ socket. Provides a *request()* method with optional timeout.

Parameters
- **context** (*zmq.Context*) -- the ZeroMQ context to create the socket in.
- **endpoint** (*str*) -- the endpoint to connect to.
- **service** (*str*) -- the service the client should use

on_message (*msg*)
Public method called when a message arrived.

Note Does nothing. Should be overloaded!

on_timeout ( )
Public method called when a timeout occurred.

Note Does nothing. Should be overloaded!

request (*msg*, *timeout=None*)
Send the given message.

Parameters
- **msg** (*list of str*) -- message parts to send.
- **timeout** (*int*) -- time to wait in milliseconds.

Rtype None

shutdown ( )
Method to deactivate the client connection completely.
Will delete the stream and the underlying socket.

Warning The instance MUST not be used after *shutdown()* has been called.

Return type None

exception dedalus.mnclient.**RequestTimeout**
Exception raised when the request timed out.

3.1 Dedalus client

class dedalus.mnclient_runner.**ClientRunner** (*context*, *endpoint*, *service*)
This class inherits *MNClient* and performs a number of tests on the performance of the broker and worker instances it is connected to.

Dedalus hardware scanner

class dedalus.scanner_hw.**HWScanner**

Provides functionality that has to do with the physical capabilities of the nodes' hardware. It can query the underlying hardware for statistics, reboot the host, control sensors and any other functionality that has to do with the hardware.

static boot_time ( )

Return the system boot time expressed in seconds since the epoch.

static children (*recursive=False*)

Return the children of this process as a list of Process objects, preemptively checking whether PID has been reused. If recursive is True return all the parent descendants.

static cmdline ( )

The command line this process has been called with as a list of strings. The return value is not cached because the cmdline of a process may change.

static connections (*kind='inet'*)

Return socket connections opened by process as a list of named tuples.

static cpu_count (*logical=False*)

Return the number of logical CPUs in the system (same as os.cpu_count() in Python 3.4). If logical is False return the number of physical cores only (hyper thread CPUs are excluded). Return None if undetermined.

static cpu_freq ( )

Return CPU frequency as a nameduple including current, min and max frequencies expressed in Mhz. On Linux current frequency reports the real-time value, on all other platforms it represents the nominal "fixed" value. If percpu is True and the system supports per-cpu frequency retrieval (Linux only) a list of frequencies is returned for each CPU, if not, a list with a single element is returned. If min and max cannot be determined they are set to 0.

static cpu_percent (*interval=1*, *percpu=True*)

Return a float representing the current system-wide CPU utilization as a percentage. When interval is > 0.0 compares system CPU times elapsed before and after the interval (blocking). When interval is 0.0 or None compares system CPU times elapsed since last call or module import, returning immediately. That means the first time this is called it will return a meaningless 0.0 value which you are supposed to ignore. In this case it is recommended for accuracy that this function be called with at least 0.1 seconds between calls. When percpu is True returns a list of floats representing the utilization as a percentage for each CPU. First element of the list

refers to first CPU, second element to second CPU and so on. The order of the list is consistent across calls.

static cpu_stats ( )
Return various CPU statistics as a named tuple

static cpu_times ( )
Return system CPU times as a named tuple. Every attribute represents the seconds the CPU has spent in the given mode. The attributes availability varies depending on the platform.

static cpu_times_percent (*interval=1*, *percpu=False*)
Same as cpu_percent() but provides utilization percentages for each specific CPU time as is returned by psutil.cpu_times(percpu=True). interval and percpu arguments have the same meaning as in cpu_percent().

static create_time ( )
The process creation time as a floating point number expressed in seconds since the epoch, in UTC. The return value is cached after first call.

static cwd ( )
The process current working directory as an absolute path.

static disk_io_counters (*perdisk=True*)
Return system-wide disk I/O statistics as a named tuple

static disk_partitions ( )
Return all mounted disk partitions as a list of named tuples including device, mount point and filesystem type, similarly to “df” command on UNIX. If all parameter is False it tries to distinguish and return physical devices only (e.g. hard disks, cd-rom drives, USB keys) and ignore all others (e.g. memory partitions such as /dev/shm). Note that this may not be fully reliable on all systems (e.g. on BSD this parameter is ignored). Named tuple’s fstype field is a string which varies depending on the platform. On Linux it can be one of the values found in /proc/filesystems (e.g. 'ext3' for an ext3 hard drive o 'iso9660' for the CD-ROM drive). On Windows it is determined via GetDriveType and can be either "removable", "fixed", "remote", "cdrom", "unmounted" or "ramdisk". On OSX and BSD it is retrieved via getfsstat(2).

static disk_usage (*directory='/'*)
Return disk usage statistics about the given path as a named tuple including total, used and free space expressed in bytes, plus the percentage usage. OSError is raised if path does not exist.

dump ( )
Return system info in bulk.

Return type dict

static echo (*echo_message*)
Return echo message

static environ ( )
The environment variables of the process as a dict. Note: this might not reflect changes made after the process started.

static exe ( )
The process executable as an absolute path. On some systems this may also be an empty string. The return value is cached after first call.

get_interface ( )
Returns the current interface used by the routing protocol.

Return type str

static gids ( )
The real, effective and saved group ids of this process as a named tuple. This is the same as os.getresgid() but can be used for any process PID.

static io_counters ( )
Return process I/O statistics as a named tuple.

static is_running (*pid*)
Return whether the current process is running in the current process list. This is reliable also in case the process is gone and its PID reused by another process.

static kill_process (*pid*)
Kill the current process by using SIGKILL signal preemptively checking whether PID has been reused.

static memory_info ( )
Return a named tuple with variable fields depending on the platform representing memory information about the process. The "portable" fields available on all platforms are rss and vms. All numbers are expressed in bytes.

static memory_maps (*grouped=True*)
Return process's mapped memory regions as a list of named tuples whose fields are variable depending on the platform. This method is useful to obtain a detailed representation of process memory usage as explained here (the most important value is "private" memory). If grouped is True the mapped regions with the same path are grouped together and the different memory fields are summed. If grouped is False each mapped region is shown as a single entity and the named tuple will also include the mapped region's address space (addr) and permission set (perms).

static memory_percent (*memtype='rss'*)
Compare process memory to total physical system memory and calculate process memory utilization as a percentage. memtype argument is a string that dictates what type of process memory you want to compare against.

static name ( )
The process name. On Windows the return value is cached after first call. Not on POSIX because the process name may change.

static net_connections ( )
Return system-wide socket connections as a list of named tuples.

static net_if_addrs ( )
Return the addresses associated to each NIC (network interface card) installed on the system as a dictionary whose keys are the NIC names and value is a list of named tuples for each address assigned to the NIC.

static net_if_stats ( )
Return information about each NIC (network interface card) installed on the system as a dictionary whose keys are the NIC names and value is a named tuple.

static net_io_counters (*pernic=True*)
Return system-wide network I/O statistics as a named tuple

static nice (*value=None*)
Get or set process niceness (priority). On UNIX this is a number which usually goes from -20 to 20. The higher the nice value, the lower the priority of the process.

static num_ctx_switches ( )
The number voluntary and involuntary context switches performed by this process (cumulative).

static num_fds ( )
The number of file descriptors currently opened by this process (non cumulative).

static num_handles ( )
The number of handles currently used by this process (non cumulative).

static num_threads ( )
The number of threads currently used by this process (non cumulative).

static open_files ( )
Return regular files opened by process as a list of named tuples.

static pid ( )
The process PID. This is the only (read-only) attribute of the class.

static pid_exists (*pid*)
Check whether the given PID exists in the current process list. This is faster than doing pid in psutil.pids() and should be preferred.

static pids ( )
Return a list of current running PIDs. To iterate over all processes and avoid race conditions process_iter() should be preferred.

static ppid ( )
The process parent PID. On Windows the return value is cached after first call. Not on POSIX because ppid may change if process becomes a zombie.

static process_cpu_num (*pid*)
Return what CPU this process is currently running on. The returned number should be <= psutil.cpu_count(). It may be used in conjunction with psutil.cpu_percent(percpu=True) to observe the system workload distributed across multiple CPUs.

static process_cpu_percent (*pid*, *interval=None*)
Return a float representing the process CPU utilization as a percentage which can also be > 100.0 in case of a process running multiple threads on different CPUs. When interval is > 0.0 compares process times to system CPU times elapsed before and after the interval (blocking). When interval is 0.0 or None compares process times to system CPU times elapsed since last call, returning immediately. That means the first time this is called it will return a meaningless 0.0 value which you are supposed to ignore. In this case is recommended for accuracy that this function be called a second time with at least 0.1 seconds between calls.

static process_cpu_times (*pid*)
Return a (user, system, children_user, children_system) named tuple representing the accumulated process time, in seconds (see explanation). On Windows and OSX only user and system

are filled, the others are set to 0. This is similar to os.times() but can be used for any process PID.

static restart_host ( )
Reboot the host machine that is running this worker.

Returns None

static resume_process (*pid*)
Resume process execution with SIGCONT signal preemptively checking whether PID has been reused.

run (*cmd*, *args*)
Will run any of the available functions in the module and return its output.
Will return None if it fails.

Parameters
- **cmd** (*bytearray*) -- the opcode of the dunction to run
- **args** (*list of str*) -- any additional arguments to pass along

Returns dict

static sensors_battery ( )
Return battery status information as a named tuple including the following values. If no battery is installed or metrics can't be determined None is returned.

static sensors_fans ( )
Return hardware fans speed. Each entry is a named tuple representing a certain hardware sensor fan. Fan speed is expressed in RPM (rounds per minute). If sensors are not supported by the OS an empty dict is returned.

static sensors_temperatures (*fahrenheit=False*)
Return hardware temperatures. Each entry is a named tuple representing a certain hardware temperature sensor (it may be a CPU, an hard disk or something else, depending on the OS and its configuration). All temperatures are expressed in celsius unless fahrenheit is set to True. If sensors are not supported by the OS an empty dict is returned.

static shutdown ( )
Close the hardware scanner.

Returns None

static status ( )
The current process status as a string.

static suspend_process (*pid*)
Suspend process execution with SIGSTOP signal preemptively checking whether PID has been reused.

static swap_memory ( )
Return system swap memory statistics as a named tuple

static terminal ( )
The terminal associated with this process, if any, else None. This is similar to "tty" command but can be used for any process PID.

static terminate_process (*pid*)
Terminate the process with SIGTERM signal preemptively checking whether PID has been

reused.

static threads ( )
Return threads opened by process as a list of named tuples including thread id and thread CPU times (user/system).

static traffic_per_connection (*sample_interval=1*, *hosts_limit=10*)
Get statistics about traffic from this node to all its' open connections.

Parameters
- **sample_interval** (*int*) -- the interval to capture traffic from an interface in seconds
- **hosts_limit** (*int*) -- the maximum number of open connections to track

Returns dict

static uids ( )
The real, effective and saved user ids of this process as a named tuple. This is the same as os.getresuid() but can be used for any process PID.

static username ( )
The name of the user that owns the process. On UNIX this is calculated by using real process uid.

static users ( )
Return users currently connected on the system as a list of named tuples.

static virtual_memory ( )
Return statistics about system memory usage as a named tuple including the following fields, expressed in bytes.

static wait_process (*pid*, *timeout=None*)
Wait for process termination and if the process is a children of the current one also return the exit code, else None. On Windows there's no such limitation (exit code is always returned). If the process is already terminated immediately return None instead of raising NoSuchProcess. If timeout is specified and process is still alive raise TimeoutExpired exception. It can also be used in a non-blocking fashion by specifying timeout=0 in which case it will either return immediately or raise TimeoutExpired.

class dedalus.scanner_hw.**HWTraffic**
Provides functionality that has to do with networking. This includes generating traffic and measuring bandwidth.

static change_txpower (*txpower='31'*, *iface='wlp3s0'*)
Change the transmission power of the currently used network interface.

Parameters
- **txpower** (*str*) -- the new level of power in dB
- **iface** (*str*) -- the name of one of the available interfaces

Returns bool

kill_throughput_server ( )
Close the throughput server if one is open.

Returns Nome

load_dtn_module ( )
Load the DTN controller if there exists one.

Returns None

load_icn_module ( )
Load the ICN controller if there exists one.

Returns None

measure_throughput (*host*, *port=50042*, *period=10*)
Measure the throughput in the link from this worker and a neighbour.

Parameters
- **host** (*str*) -- the address of a neighbour to connect to
- **port** (*int*) -- the port to use for the measurement
- **period** (*int*) -- the time to run the measurement for

Returns dict

open_throughput_server ( )
Start a througput server in a separate thread and store its PID.

Returns Nome

static **ping_test** (*addr*, *n=10*)
Will ping a node and record statistics.
If an error occurred during execution then a message will be returned instead.

Parameters
- **addr** (*str*) -- the address of the node to ping
- **n** (*int*) -- the number of ping requests to send

Returns dict

run (*cmd*, *arg*)
This is an input point for running all available functions in this module.

Parameters
- **cmd** (*bytearray*) -- the opcode for the requested functionality
- **arg** (*list of str*) -- any other arguments to pass along

Returns dict

search_icn_file (*node*)
Send an ICN request for a file.
Will return an error message if it fails

Parameters **node** (*str*) -- the ICN tag of a node

Returns bool

send_dtn_file (*destination*)
Send an autogenerated file to a node.
Will return an error message if it fails

Parameters **destination** (*str*) -- the DTN tag of a node

Returns bool

shutdown ( )
Will shutdown the traffic controller.

Returns None

static throughput_listener (*port=50042*)
A server to listen for throughput requests from a neighbour.

Parameters **port** (*int*) -- the port to bind to

Returns None

static traceroute_test (*addr*)
Return the currently active path to a node.

Parameters **addr** (*str*) -- the address of the node to connect to

Returns dict

class dedalus.scanner_hw.**NetworkManager**
Provides a consistent API for opening or closing all available protocol modules.
Every network protocol supported by Dedalus is added here.

change_dtn_protocol (*protocol*)
Will change the active DTN protocol.
Will close the currently running protocol if any and then start the selected one if it is available.
Will return True if it succeeds.

Returns bool

change_protocol (*protocol*)
Change a routing protocol.
Will close the currently running protocol if any and then start the selected one if it is available.
Will return True if it succeeds.

Parameters **protocol** (*str*) -- the new protocol that will be opened

Returns bool

find_available_protocols ( )
Check which of the registered protocol modules are available in the system.
Once this function is run, the current instance of the NetworkManager will be automatically notified.

Returns None

get_ccn_dump ( )
Get all available information from the currently active ICN protocol.
Will return None if it fails.

Returns dict

get_current_dtn_protocol ( )
Get the currently active DTN protocol name.

Returns str

get_current_routing_protocol ( )
Will return the name of the currently active routing protocol.

Returns str

get_dtn_dump ( )
Get all available information from the currently active DTN protocol.
Will return None if it fails.

Returns dict

get_routing_dump ( )
Return all available information regarding the routing module.
Will return all available neighbours for this node, link costs, as well as routing rules.

Returns dict

import_scanner_module ( )
Import the controllers for every available protocol module

Returns None

load_dtn_module ( )
Load the DTN controller.

Returns None

load_icn_module ( )
Load the controller for the ICN module.
Will return True if it succeeds.

Returns dict

restart_dtn ( )
Restart the currently running DTN protocol.
Will return True if it succeeds.

Returns bool

restart_icn ( )
Restart the currently running ICN protocol.
Will return True if it succeeds.

Returns bool

restart_protocol ( )
Restart the currently running routing protocol.
Will return True if it succeeds.

Returns bool

run (*cmd*, *arg*)
Execute a command.

Parameters
- **cmd** (*bytearray*) -- the opcode for the command to be executed
- **arg** (*list of str*) -- any additional arguments to be passed along

Returns None

shutdown ( )
Will shut down all active protocols.

Returns None

start_dtn ( )
Will start the selected DTN protocol.
Will return True if it succeeds.

Returns bool

start_icn ( )
Will start the selected ICN protocol.
Will return True if it succeeds.

Returns bool

start_network ( )
Start the currently selected routing protocol.
Will return True if it succeeds.

Returns bool

stop_dtn ( )
Will stop the currently running DTN protocol.
Will return True if it succeeds.

Returns bool

stop_icn ( )
Will stop the currently running ICN protocol.
Will return True if it succeeds.

Returns bool

stop_network ( )
Close the currently running routing protocol.
Will return True if it succeeds.

Returns bool

CHAPTER 5

Dedalus controllers

5.1 Ad Hoc

5.1.1 BABEL

class dedalus.protocol_modules.scanner_babel.**BabelNode**

Represents a node that runs the bmx7 routing protocol. It will keep track of the routing status and neighbour information.

neighbors ( )

Will reeturns the neighbours list for this node

Return type list

routing_info ( )

Collect and present all the related routing infromation for this node. This involves communication interface status, neighbours and link states.

Return type dict

class dedalus.protocol_modules.scanner_babel.**Scanner** (*ip='::1', port=33123, buffer_size=1024*)

Will scan and collect information from the BMX routing protocol.

Parameters

- **ip** (*str*) -- the ip used to establish the TCP connection with babel daemon
- **port** (*int*) -- the port used to establish the TCP connection with babel daemon
- **buffer_size** (*int*) -- the buffer used to receive the babel reply

get_current_interface ( )

Will figure out the interface used for routing and store it in the Node class.

get_node_info ( )

Method that returns the babel node.

Return type *BabelNode*

get_routing_info ( )

Return a data dump of all the currently stored routing information.

Return type dict

static recvall (*s*, *buffer_size*, *termination_chars=None*)
Method called to receive Babel's reply for any particular call.

Note if there are no termination characters then we read once and return.

Note if there are we read until we stumble into one.

Parameters
- **s** (*Socket*) -- the socket to use for the connection
- **buffer_size** (*int*) -- the size of the buffer used
- **termination_chars** (*list of str*) -- characters that indicate the termination of the answer

Return type str

run_cmd (*msg_code*)
Will make the appropriate function call demending on the code provided.

Parameters msg_code (*str*) -- The code of the function to execute

start ( )
Start the scanner.
Will immediately start collecting all available information about the current routing status and store it in the *BabelNode* structures.

Note Will only return a string message in the case that it detects another Scanner instance running.

---

static stop ( )
Stop the scanner.
Will close the currently open Babel protocol instance and clear the Scanner module.

Return type str

5.1.2 BMX7

class dedalus.protocol_modules.scanner_bmx.**BMXNode**
Represents a node that runs the bmx7 routing protocol. It will keep track of the routing status and neighbour information.

neighbors ( )
Will reeturns the neighbours list for this node

Return type list

routing_info ( )
Collect and present all the related routing infromation for this node. This involves communication interface status, neighbours and link states.

Return type dict

class dedalus.protocol_modules.scanner_bmx.**Scanner**
Will scan and collect information from the BMX routing protocol.

get_current_interface ( )
Will figure out the interface used for routing and store it in the Node class.

get_routing_info ( )
Return a data dump of all the currently stored routing information.

Return type dict

start ( )
Start the scanner.
Will immediately start collecting all available information about the current routing status and store it in the *BMXNode* structures.

Note Will only return a string message in the case that it detects another Scanner instance running.

---

static stop ( )
Stop the scanner.
Will close the currently open BMX protocol instance and clear the Scanner module.

Return type str

5.2 DTN

5.2.1 IBR

class dedalus.protocol_modules.scanner_ibr.**IBRNode**
Represents a node that runs the IBR routing protocol. It will keep track of the routing status and neighbour information.

neighbors ( )
Will reeturns the neighbours list for this node

Return type list

routing_info ( )
Collect and present all the related dtn routing infromation for this node. This involves communication interface status, neighbours and bundle information.

Return type dict

class dedalus.protocol_modules.scanner_ibr.**Scanner** (*ip='::1', port=4550, buffer_size=1024*)
Will scan and collect information from the BMX routing protocol.

Parameters
- **ip** (*str*) -- the ip used to establish the TCP connection with babel daemon
- **port** (*int*) -- the port used to establish the TCP connection with babel daemon
- **buffer_size** (*int*) -- the buffer used to receive the babel reply

static **get_current_interface** ( )
Will figure out the interface used for routing and store it in the Node class.

get_name ( )
Will return the unique node name used for all DTN communications.

Return type str

get_node ( )
Method that returns the IBR node.

Return type *IBRNode*

get_routing ( )
Will return all DTN available routes known to this node.

Return type dict

get_routing_info ( )
Return a data dump of all the currently stored routing information.

Return type dict

ping (*dtn_node*)
Will ping a DTN node using bundles.

Note Results are saved in the Log directory of Dedalus

Parameters **dtn_receiver** (*int*) -- the name of a DTN node

static **recvall** (*fs*, *status_codes*)
Get a reply from the DTN protocol running in this system.

Parameters
- **fs** (*Socket*) -- the socket that has been opened for connecting with the protocol.
- **status_codes** (*dic*) -- all possible status codes of the protocol

Return type str

send_files (*dtn_receiver*)
Will send a file using the currently running DTN protocol.

Note For now this is an automatically generated file containing origin and timestamp in its name.

Parameters **dtn_receiver** (*int*) -- the name of a DTN node

send_traffic (*MB*, *dtn_receiver*)
Will transmit bundles equal to a given amount in BM to a given DTN node.

> **Warning** This function is not used for now.

Parameters
- **MB** -- the size of the total data to be transmitted.
- **dtn_receiver** (*int*) -- the name of a DTN node

Typr MB int

send_traffic_until (*minutes*, *dtn_receiver*)
Will transmit bundles for a given amount of time to a given DTN node.

> **Warning** This function is not used for now.

Parameters
- **minutes** -- the amount of time to transmit in minutes.
- **dtn_receiver** (*int*) -- the name of a DTN node

Typr minutes int

start ( )
Start the scanner.
Will immediately start collecting all available information about the current dtn routing status and store it in the *IBRNode* structures.

> **Note** Will only return a string message in the case that it detects another Scanner

static stop ( )
Stop the scanner.
Will close the currently open DTN protocol instance and clear the Scanner module.

Return type str

5.3 ICN

5.3.1 CCN-Lite

class dedalus.protocol_modules.scanner_ccn.**CCNNode**
Represents a node that runs a CCN routing protocol. It will keep track of the routing status and neighbour information.

info ( )
Collect and present all the related routing infromation for this node. This includes the protocol state for now.

Return type dict

class dedalus.protocol_modules.scanner_ccn.**Scanner**
Will scan and collect information from the CCN protocol and provide an interface to its available functionalities like file requests and advertisements.

add_face (*address*, *node*)
Will create a new face based on the address and the node ID given.

Parameters
- **address** (*str*) -- IP address of the CCN node to add.
- **node** (*str*) -- ID of the node as returned by *read_face()*

create_files ( )
Will create a number of default files that will be advertised by this node.

delete_face (*node*)
Will delete the face of a record associated with a particular node ID.
..note::It will leave the node address intact.

delete_sockets ( )
Will close the CCN relays used by this node.

Return type str

get_local_address ( )
Will return the address used by this node for CCN related operations.

Return type str

get_neighbours_route ( )
Get the routes to the available neighbours that support a CCN routing protocol.
Will add each neighbour as a new record in the scanner.

get_routing_info ( )
Collect and present all the related ccn routing infromation for this node. This involves node names for now.

Return type dict

neighbours_background ( )
Will scan for neighbour changes and update the appropriate records of the scanner.

read_face ( )
Will read a face ID from a CCN-Lite information dump and return it.

Return type str

restart_server ( )
Stop and re-start the protocols and modules involved in the CCN routing and functionality of the node.

Return type dict

search (*node='node1'*)
Convenience wrapper around *search_content()*.
Will return a dictionary with a 'success' status and any other relevant information.

Return type dict

search_content (*local, path*)
Will create a request for a file and forward it to the CCN network.

Parameters
- **local** (*str*) -- the IP address of the node to forward this request.
- **path** (*str*) -- the address of the file we want to retrieve

Return type str

start ( )
Start the scanner.

Will immediately start the CCN routing protocol.
Will return a dictionary with a success status and relevant information.

Return type dict

stop ( )
Stop the scanner.
Will close the currently open CCN protocol instance and clear the Scanner module.

Return type dict

- *Index*
- *Module Index*
- *Search Page*

d

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W