RIAPS Documentation

RIAPS Team

Jan 04, 2024

CONTENTS:

This documents the APIs for RIAPS application management (riaps_ctrl and riaps_deplo), Python component framework (riaps_actor and riaps_device), node management tool (riaps_fab using fabric), code generation tool (riaps_gen) and an alternate discovery service (redis-based).

CHAPTER

ONE

RIAPS PACKAGE

1.1 Subpackages

1.1.1 riaps.consts package

Submodules

riaps.consts.const module

Const class to emulate constants.

Created on Oct 23, 2016

@author: riaps

class riaps.consts.const.**const**

Bases: object

exception ConstError

Bases: TypeError

riaps.consts.defs module

Constants for the run-time system Created on Oct 20, 2016

@author: riaps

Module contents

1.1.2 riaps.ctrl package

Submodules

riaps.ctrl.ctrl module

riaps.ctrl.ctrlcli module

riaps.ctrl.ctrlgui module

riaps.ctrl.ctrlsrv module

riaps.ctrl.main module

Module contents

1.1.3 riaps.deplo package

Submodules

riaps.deplo.appdb module

Application database Created on Apr 2, 2018

@author: riaps

class riaps.deplo.appdb.**AppDbase**

Bases: object

Application database. The database is a collection of key -> value pairs, where values are pickled Python objects (i.e. bytearrays). Structure: RIAPSAPPS -> [appNames*] appName -> [actorRecords*]

RIAPSAPPS = 'RIAPSAPPS'

RIAPSDISCO = 'RIAPSDISCO'

RIAPSDISCOCMD = 'RIAPSDISCOCMD'

addApp(*appName*)

Add a new app to the database

addAppActor(*appName*, *actorRecord*)

closeDbase()

delApp(*appName*)

delAppActor(*appName*, *actorName*)

delDiscoCommand()

delKey(*key*)

getAppActor(*appName*, *actorName*)

getAppActors(*appName*)

getApps()

getDisco()

getDiscoCommand()

getKeyValue(*key*, *default=None*)

putKeyValue(*key*, *value*)

replaceKeyValue(*key*, *value*)

setDisco(*disco*)

setDiscoCommand(*discoCmd*)

riaps.deplo.cpumon module

Resource monitors

Created on Nov 23, 2017

@author: riaps

class riaps.deplo.cpumon.**CPUMonitorThread**(*parent*, *interval*, *usage*)

Bases: Thread

Thread for monitoring an actor and enforcing resource limits.

addClientDevice(*appName*, *actorName*, *device*)

is_running()

restart()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setup(*proc*)

stop()

terminate()

riaps.deplo.deplo module

riaps.deplo.depm module

riaps.deplo.fm module

riaps.deplo.main module

riaps.deplo.memmon module

Resource monitors

Created on Nov 23, 2017

@author: riaps

class riaps.deplo.memmon.**MemMonitorThread**(*parent*, *efd*, *usage*)

Bases: Thread

addClientDevice(*appName*, *actorName*, *device*)

is_running()

restart()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setup(*proc*)

stop()

terminate()

riaps.deplo.netmon module

Resource monitors

Created on Nov 23, 2017

@author: riaps

class riaps.deplo.netmon.**NHAction**

Bases: object

MAP = {1: 'SET', 2: 'REMOVE'}

REMOVE = 2

SET = 1

class riaps.deplo.netmon.**NHLoopStatus**

Bases: object

Return codes from nethogsmonitor_loop()

FAILURE = 1

MAP = {0: 'OK', 1: 'FAILURE', 2: 'NO_DEVICE'}

NO_DEVICE = 2

OK = 0

class riaps.deplo.netmon.**NHMonitorRecord**

Bases: Structure

ctypes version of the struct of the same name from libnethogs.h

device_name

Structure/Union member

name

Structure/Union member

pid

Structure/Union member

record_id

Structure/Union member

recv_bytes

Structure/Union member

recv_kbs

Structure/Union member

sent_bytes

Structure/Union member

sent_kbs

Structure/Union member

uid

Structure/Union member

class riaps.deplo.netmon.**NetMonitorThread**(*parent*)

Bases: Thread

addClientDevice(*appName*, *actorName*, *device*, *proc*, *rate*)

addProc(*appName*, *actName*, *proc*)

delProc(*appName*, *actName*, *proc*)

dev_args(*devnames*)

Return the appropriate ctypes arguments for a device name list, to pass to libnethogs `nethogsmonitor_loop_devices`. The return value is a 2-tuple of devc (`ctypes.c_int`) and devicenames (`ctypes.POINTER`) to an array of `ctypes.c_char`).

Parameters
devnames (*list*) – list of device names to monitor

Returns
2-tuple of devc, devicenames ctypes arguments

Return type
tuple

is_running()

network_activity_callback(*_action*, *data*)

restart()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()

terminate()

riaps.deplo.procm module

Created on Apr 7, 2018

@author: riaps

class riaps.deplo.procm.**ProcessManager**(*parent*)

Bases: object

Manages processes: service(s) and actors started by deplo

monitor(*qualName*, *proc*)

release(*qualName*)

class riaps.deplo.procm.**ProcessMonitor**(*parent*, *qualName*)

Bases: Thread

Thread for monitoring a process

error()

release()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

setup(*record*)

terminate()

class riaps.deplo.procm.**ProcessMonitorRecord**(*name*, *proc*, *thread*)

Bases: tuple

name

Alias for field number 0

proc

Alias for field number 1

thread

Alias for field number 2

riaps.deplo.resm module

Resource manager

Created on Nov 23, 2017

@author: riaps

class riaps.deplo.resm.**ActorResourceManager**(*parent*, *actorName*, *actorDef*)

Bases: object

Resource manager for an actor

addClientDevice(*device*)

cleanupActor()

setupCPU()

setupMem()

setupNet()

setupSpace()

startActor(*proc*)

stopActor(*proc*)

class riaps.deplo.resm.**AppResourceManager**(*parent*, *appName*, *appFolder*, *userName*)

Bases: object

Resource manager for an app

addActor(*actorName*, *actorDef*)

addClientDevice(*actorName*, *device*)

addQuota(*usage*)

claimApp()

cleanupApp()

delQuota()

getUserName()

nextActorID()

reclaimApp()

startActor(*actorName*, *proc*)

stopActor(*actorName*, *proc*)

class riaps.deplo.resm.**ResourceManager**(*context*)

Bases: object

Resource manager

addActor(*appName*, *actorName*, *actorDef*)

addClientDevice(*appName*, *actorName*, *device*)

cleanupApp(*appName*)

cleanupApps()

Cleanup all apps: remove the cgroup of the apps

cleanupNet()

getUserName(*appName*)

reclaimApp(*appName*)

setupApp(*appName*, *appFolder*, *userName*)

Start an app: create an app resource manager for this app if it has not been created yet

startActor(*appName*, *actorName*, *proc*)

stopActor(*appName*, *actorName*, *proc*)

terminate()

riaps.deplo.spcmon module

Resource monitors

Created on Nov 23, 2017

@author: riaps

class riaps.deplo.spcmon.**SpcMonitorThread**(*parent*)

Bases: Thread

addClientDevice(*appName*, *actorName*, *device*, *proc*)

addProc(*appName*, *actName*, *proc*)

delProc(*appName*, *actName*, *proc*)

is_running()

restart()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop()

terminate()

Module contents

1.1.4 riaps.discd package

Submodules

riaps.discd.dbase module

Discovery server database interface Created on Oct 19, 2016

@author: riaps

class riaps.discd.dbase.**DiscoDbase**(*context_*, *dbaseLoc*)

Bases: object

Discovery service database base class

detach(*key: str*, *target: str*)

Detach client (for updates) from keys

fetch(*key: str*, *client: str*) → [<class 'str'>]

Fetch value(s) under key. Add client to list of clients interested in the value

fetchUpdates() → [<class 'str'>]

Check and fetch the updated values of the subscribed keys if any

insert(*key: str*, *value: str*) → [<class 'str'>]

Insert value under key and return list of clients of value (if any). A key may have multiple values associated with it, hence the new value is added to the set of values that belong to the key

remove(*key: str*, *value: str*) → [<class 'str'>]

Remove value from value under key.

start()

Start the database: connect to the database process

terminate()

riaps.discd.dbase_dht module

riaps.discd.dbase_redis module

Discovery server database interface Created on Oct 19, 2016

@author: riaps

class riaps.discd.dbase_redis.**RedisDbase**(*context_*, *dbaseLoc*)

Bases: *DiscoDbase*

Discovery service database implemented using redis

addSub(*newKey*)

Update the list of subscribed keys with the new key

delSub(*key*)

Delete subscription to key

detach(*key: str*, *target: str*)

Detach update client from keys

fetch(*key: str*, *client: str*) → [<class 'str'>]

Fetch value(s) under key. Add client to list of clients interested in the value

fetchUpdates()

Check and fetch the updated values of the subscribed keys if any

insert(*key: str*, *value: str*) → [<class 'str'>]

Insert value under key and return list of clients of value (if any). A key may have multiple values associated with it, hence the new value is added to the set of values that belong to the key

remove(*key: str*, *value: str*) → [<class 'str'>]
Remove value from values under key.

start()
Start the database: connect to the database process

riaps.discd.discs module

riaps.discd.main module

Module contents

1.1.5 riaps.fabfile package

Submodules

riaps.fabfile.deplo module

riaps.fabfile.fabfile module

riaps.fabfile.riaps module

riaps.fabfile.sys module

riaps.fabfile.time module

Module contents

1.1.6 riaps.gen package

Subpackages

riaps.gen.target package

Subpackages

riaps.gen.target.capnp package

Submodules

riaps.gen.target.capnp.capnpfilters module

riaps.gen.target.capnp.capnpfilters.**generate_capnp_id**(*value*)

riaps.gen.target.capnp.capnpgen module

class riaps.gen.target.capnp.capnpgen.**CapnpGenerator**(*cppmodel*, *output_dir*)

Bases: JinjaGenerator

create_environment(***kwargs*)

Return a new Jinja environment.

Derived classes may override method to pass additional parameters or to change the template loader type.

tasks = [<riaps.gen.target.capnp.capnpgen.CapnpTask object>]

templates_path = '/home/docs/checkouts/readthedocs.org/user_builds/riaps-pycom/checkouts/latest/src/riaps/gen/target/capnp/tpl'

class riaps.gen.target.capnp.capnpgen.**CapnpTask**(*formatter=None*, ***kwargs*)

Bases: JinjaTask

filtered_elements(*model*)

Iterator over model elements to execute this task for.

relative_path_for_element(*element*)

Returns relative file path receiving the generator output for given element.

template_name = 'capnp.tpl'

riaps.gen.target.capnp.sync_capnp module

class riaps.gen.target.capnp.sync_capnp.**FileSync**(*model*)

Bases: object

apply_capnp_rules(*orig_filepath*, *new_filepath*)

sync_capnp(*output_dir*)

Module contents

riaps.gen.target.cpp package

Submodules

riaps.gen.target.cpp.ccfilters module

riaps.gen.target.cpp.ccfilters.**cpp_port_type**(*port_type*)

riaps.gen.target.cpp.ccfilters.**handler_name**(*value*)

riaps.gen.target.cpp.ccfilters.**port_macro**(*value*, *port_type*)

riaps.gen.target.cpp.ccfilters.**recv_message_type**(*value*, *port_type*)

riaps.gen.target.cpp.ccfilters.**recv_return_type**(*value*, *port_type*)

riaps.gen.target.cpp.ccfilters.**sender_message_type**(*value*, *port_type*)

riaps.gen.target.cpp.ccfilters.**sender_name**(*value*)

riaps.gen.target.cpp.cppgen module

class riaps.gen.target.cpp.cppgen.**CmakeTask**(*formatter=None*, ***kwargs*)

Bases: JinjaTask

filtered_elements(*model*)

Iterator over model elements to execute this task for.

relative_path_for_element(*element*)

Returns relative file path receiving the generator output for given element.

template_name = 'cmake.tpl'

class riaps.gen.target.cpp.cppgen.**CompCppBaseTask**(*part*)

Bases: JinjaTask

filtered_elements(*model*)

Iterator over model elements to execute this task for.

relative_path_for_element(*element*)

Returns relative file path receiving the generator output for given element.

template_name = 'comp.base.cc.tpl'

class riaps.gen.target.cpp.cppgen.**CompCppTask**(*part*)

Bases: JinjaTask

filtered_elements(*model*)

Iterator over model elements to execute this task for.

relative_path_for_element(*element*)

Returns relative file path receiving the generator output for given element.

template_name = 'comp.cc.tpl'

class riaps.gen.target.cpp.cppgen.**CompGenerator**(*environment=None*, ***kwargs*)

Bases: JinjaGenerator

create_environment(***kwargs*)

Return a new Jinja environment.

Derived classes may override method to pass additional parameters or to change the template loader type.

tasks = [<riaps.gen.target.cpp.cppgen.CompHppBaseTask object>,
<riaps.gen.target.cpp.cppgen.CompCppBaseTask object>,
<riaps.gen.target.cpp.cppgen.CompHppTask object>,
<riaps.gen.target.cpp.cppgen.CompCppTask object>,
<riaps.gen.target.cpp.cppgen.CompHppBaseTask object>,
<riaps.gen.target.cpp.cppgen.CompCppBaseTask object>,
<riaps.gen.target.cpp.cppgen.CompHppTask object>,
<riaps.gen.target.cpp.cppgen.CompCppTask object>,
<riaps.gen.target.cpp.cppgen.CmakeTask object>]

templates_path = '/home/docs/checkouts/readthedocs.org/user_builds/riaps-pycom/checkouts/latest/src/riaps/gen/target/cpp/tpl'

class riaps.gen.target.cpp.cppgen.CompHppBaseTask(*part*)

Bases: JinjaTask

filtered_elements(*model*)

Iterator over model elements to execute this task for.

relative_path_for_element(*element*)

Returns relative file path receiving the generator output for given element.

template_name = 'comp.base.h.tpl'

class riaps.gen.target.cpp.cppgen.CompHppTask(*part*)

Bases: JinjaTask

filtered_elements(*model*)

Iterator over model elements to execute this task for.

relative_path_for_element(*element*)

Returns relative file path receiving the generator output for given element.

template_name = 'comp.h.tpl'

riaps.gen.target.cpp.sync_cpp module

class riaps.gen.target.cpp.sync_cpp.FileSync(*model*)

Bases: object

apply_cmake_rules(*orig_filepath*, *new_filepath*)

apply_cpp_rules(*orig_filepath*, *new_filepath*)

sync_all(*output_dir*)

sync_cmake(*output_dir*)

sync_code(*output_dir*)

Module contents

riaps.gen.target.python package

Submodules

riaps.gen.target.python.pygen module

class riaps.gen.target.python.pygen.CompGenerator

Bases: JinjaGenerator

create_environment(***kwargs*)

Return a new Jinja environment.

Derived classes may override method to pass additional parameters or to change the template loader type.

class riaps.gen.target.python.pygen.**CompPyTask**(*part*)

Bases: JinjaTask

filtered_elements(*model*)

Iterator over model elements to execute this task for.

relative_path_for_element(*element*)

Returns relative file path receiving the generator output for given element.

template_name = 'comp.py.tpl'

riaps.gen.target.python.sync_python module

class riaps.gen.target.python.sync_python.**FileSync**(*model*)

Bases: object

apply_py_rules(*orig_filepath*, *new_filepath*)

sync_code(*output_dir*)

Module contents

Module contents

Submodules

riaps.gen.gen module

riaps.gen.gen.**main**()

riaps.gen.gen.**preprocess**(*model*)

Module contents

1.1.7 riaps.lang package

Submodules

riaps.lang.depl module

Deployment language processor Created on Nov 7, 2016

@author: riaps

exception riaps.lang.depl.**DeplError**(*message*)

Bases: Exception

class riaps.lang.depl.**DeploymentModel**(*fileName*, *debug=False*, *verbose=False*)

Bases: object

Deployment model loader/parser

getActuals(*actuals*)

getAppName()

getDeployments()

getNetwork()

riaps.lang.depl.**main**(*standalone=False*)

riaps.lang.deplI module

Top-level script to start the deployment language processor ('depl') Created on Oct 15, 2016

Arguments:
model : Name of deployment model file to be processed

@author: riaps

riaps.lang.gviz module

Created on Mar 17, 2018

@author: riaps

riaps.lang.gviz.**cleanupMessages**(*G*, *msgMap*, *msgMapUsed*)

riaps.lang.gviz.**findMsgNode**(*msgType*, *actorLocals*, *actorLocalMessageNodes*, *actorInternals*, *actorInternalMessageNodes*, *msgMap*, *msgMapUsed*, *localMsgGraph*, *internalMsgGraph*)

riaps.lang.gviz.**findMsgNodePair**(*msgType*, *actorLocals*, *actorLocalMessageNodes*, *actorInternals*, *actorInternalMessageNodes*, *msgMap*, *localMsgGraph*, *internalMsgGraph*, *globalMsgGraph*)

riaps.lang.gviz.**gviz**(*model*, *deplo*, *verbose=False*)

riaps.lang.gviz.**main**(*debug=False*)

riaps.lang.gviz.**unique**(*name*)

riaps.lang.gviz.**visualize**(*deplo*, *models*)

riaps.lang.gviz.**visualize_actors**(*G*, *appModel*, *hostName*, *hostLabel*, *actors*, *msgMap*, *msgMapUsed*, *globalMsgSubgraph*)

riaps.lang.gviz.**visualize_messages**(*G*, *appModel*, *msgMap*, *msgMapUsed*)

riaps.lang.lang module

DSL for RIAPS software models Created on Oct 9, 2016 Uses the textX parser @author: riaps

exception riaps.lang.lang.**LangError**(*message*)

Bases: Exception

class riaps.lang.lang.**RiapsModel2JSON**(*model*)

Bases: object

Class to convert the RIAPS model (constructed by the parser) into a data structure suitable for generating JSON output.Dependent on the DSL syntax and the object structure built by the parser.

static convertMem(*value*, *unit*)

Convert all memory size values to kilobytes

convertRate(*value*, *unit*)

Convert all rate values to bytes/sec

static convertTime(*value*, *unit*)

Convert all time values to msec

getActorScheduler(*sched*)

getActors(*actors*)

getActuals(*actuals*)

getComponentScheduler(*sched*)

getComponents(*components*)

getFormals(*formals*)

getGroups(*groups*)

getIOComponents(*components*)

getImpl(*comp*)

getInstances(*instances*)

getInternals(*internals*)

getLibraries(*libraries*)

getLocals(*locals_*)

getMessages(*messages*)

getPorts(*ports*)

getUsage(*usage*)

riaps.lang.lang.**actor_obj_processor**(*actor*)

riaps.lang.lang.**compileModel**(*modelFileName*, *verbose=False*, *debug=False*, *generate=True*)

riaps.lang.lang.**insport_obj_processor**(*insport*)

riaps.lang.lang.**instance_obj_processor**(*instance*)

riaps.lang.lang.**main**(*debug=False*)

riaps.lang.lang.**op_port_obj_processor**(*port*)

riaps.lang.lang.**timed_port_obj_processor**(*port*)

riaps.lang.lang.**timport_obj_processor**(*timport*)

Module contents

1.1.8 riaps.logger package

Subpackages

riaps.logger.drivers package

Submodules

riaps.logger.drivers.base_driver module

class riaps.logger.drivers.base_driver.**BaseDriver**(*driver_type*)

Bases: ABC

close()

abstract handle(*msg*)

riaps.logger.drivers.console_driver module

class riaps.logger.drivers.console_driver.**ServerLogDriver**(*driver_type*, *session_name*)

Bases: *BaseDriver*

close()

handle(*msg*)

riaps.logger.drivers.factory module

riaps.logger.drivers.file_driver module

class riaps.logger.drivers.file_driver.**ServerLogDriver**(*driver_type*, *session_name*)

Bases: *BaseDriver*

close()

handle(*msg*)

riaps.logger.drivers.tmux_driver module

Module contents

Submodules

riaps.logger.main module

riaps.logger.riaps_log_config_test module

Script to test app log config file

Created on Oct 20, 2022

Arguments -f (or –file) FILE : Path to the file that will be used to construct the loggers @author: riaps

riaps.logger.riaps_log_config_test.**main**()

riaps.logger.riaps_log_config_test.**test_loggers**(*loggers*, *msg*)

riaps.logger.server module

Module contents

1.1.9 riaps.proto package

Module contents

1.1.10 riaps.run package

Submodules

riaps.run.actor module

riaps.run.ansPort module

Created on Oct 10, 2016

@author: riaps

class riaps.run.ansPort.**AnsPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *DuplexBindPort*

classdocs

ans_port_recv(*is_pyobj*)

ans_port_send(*msg*, *is_pyobj*)

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

`getInfo()`

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

`getSocket()`

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

`get_identity()`

`inSocket()`

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

`recv()`

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

`recv_pyobj()`

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

`reset()`

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

send(*msg*)

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

send_pyobj(*msg*)

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

set_identity(*identity*)

setup()

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

setupSocket(*owner*)

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (*Component*) – The Component the port belongs to. This operation must be called from the component thread only.

update(*host*, *port*)

Update the socket with information from the discovery service. Abstract, subclasses must implement this method.

Called when the discovery service notifies the actor about a new service provide (e.g. server, publisher, etc.) the port needs to connect to. The operation will perform the connection.

Parameters

- **host** (*str*) – IP address of the service provider
- **port** (*int*) – port number of the service provider

riaps.run.cltPort module

Client port class Created on Oct 10, 2016

@author: riaps

class riaps.run.cltPort.**CltPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *DuplexConnPort*

Client port is to access a server. Has a request and a response message type, and uses a REQ socket.

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

`getInfo()`

Retrieve relevant information about this port

`getSocket()`

Return the socket of port

`inSocket()`

Return False because the socket is not used as direct input (client has to recv explicitly)

`recv()`

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

`recv_pyobj()`

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

`reset()`

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

`send(`*msg*`)`

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

`send_pyobj(`*msg*`)`

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

`setup()`

Set up the port

setupSocket(*owner*)

Set up the socket of the port. Return a tuple suitable for querying the discovery service for the publishers

riaps.run.comp module

Component class Created on Oct 15, 2016

@author: riaps

class riaps.run.comp.**Component**

Bases: object

Base class for RIAPS application components

GROUP_PRIORITY_MAX = 0

GROUP_PRIORITY_MIN = 256

getActorID()

Return a globally unique ID (8 bytes) for the parent actor.

getActorName()

Return the name of the parent actor (as in model)

getAppName()

Return the name of the parent application (as in model)

getLocalID()

Return a locally unique ID (int) of the component. The ID is unique within the actor.

getName()

Return the name of the component (as in model)

getTypeName()

Return the name of the type of the component (as in model)

getUUID()

Return the deployment-unique ID for the parent actor

handleActionVoteRequest(*group*, *rfvId*, *when*)

Default handler for request to vote an action in the future (in member) Implementation must recv/recv_pyobj to obtain the action topic.

handleActivate()

Default activation handler

handleCPULimit()

Default handler for CPU limit exceed

handleDeactivate()

Default deactivation handler

handleDeadline(*_funcName*)

Default handler for deadline violation

handleGroupMessage(*_group*)

Default handler for group messages. Implementation must immediately call recv/recv_pyobj on the group to obtain message.

handleLeaderElected(*group*, *leaderId*)

Default handler for 'leader elected' events

handleLeaderExited(*group*, *leaderId*)

Default handler for 'leader exited' events

handleMemLimit()

Default handler for memory limit exceed

handleMemberJoined(*group*, *memberId*)

Default handler for 'member join' events

handleMemberLeft(*group*, *memberId*)

Default handler for 'member leave' events

handleMessageFromLeader(*group*)

Default handler for messages received from the leader (in member) Member implementation must immediately call recv/recv_pyobj on the group to obtain message.

handleMessageToLeader(*group*)

Default handler for messages sent to the leader (in leader) Leader implementation must immediately call recv/recv_pyobj on the group to obtain message.

handleNICStateChange(*state*)

Default handler for NIC state change

handleNetLimit()

Default handler for space limit exceed

handlePassivate()

Default activation handler

handlePeerStateChange(*state*, *uuid*)

Default handler for peer state change

handleSpcLimit()

Default handler for space limit exceed

handleVoteRequest(*group*, *rfvId*)

Default handler for vote requests (in member) Implementation must recv/recv_pyobj to obtain the topic.

handleVoteResult(*group*, *rfvId*, *vote*)

Default handler for the result of a vote (in member)

joinGroup(*groupName*, *instName*, *groupMinSize=1*, *groupPriority=256*)

leaveGroup(*group*)

class riaps.run.comp.**ComponentThread**(*parent*)

Bases: Thread

Component execution thread. Runs the component's code, and communicates with the parent actor.

addGroupSocket(*group*, *groupPriority*)

batchScheduler(*sockets*)

Batch scheduler for the component message processing.

The dictionary containing the active sockets is scanned and the associated handler is invoked.

delGroupSocket(*group*)

executeHandlerFor(*socket*)

Execute the handler for the socket

The handler is always allowed to run to completion, the operation is never preempted.

getInfo()

logEvent(*msg*)

priorityScheduler(*sockets*)

priority scheduler for the component message processing.

The priority order is determined by the order of component ports. The dictionary of active sockets is scanned, and the they are inserted into a priority queue (according to their priority value). The queue is processed (in order of priority). After each invocation, the inputs are polled (in a no-wait operation) and the priority queue is updated.

replaceSocket(*portObj*, *newSocket*)

rrScheduler(*sockets*)

Round-robin scheduler for the component message processing.

The round-robin order is determined by the order of component ports. The dictionary of active sockets is scanned, and the associated handlers are invoked in round-robin order. After each invocation, the inputs are polled (in a no-wait operation) and the round-robin queue is updated.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

runCommand()

sendControl(*msg*)

setupControl()

Create the control socket and connect it to the socket in the parent part

setupPoller()

setupScheduler()

Select the message scheduler algorithm based on the model.

setupSockets()

riaps.run.dc module

Distributed coordination

Python implementation of group formation, communications, leader election, consensus and action coordination.

Created on Feb 23, 2019 Author: riaps

class riaps.run.dc.**Coordinator**(*parent*)

Bases: object

Coordinator object. Each component has a coordinator object that creates the group objects for a component. A group instance can be created only once in a component, subsequent creations return the same group object

getGroup(*groupType*, *groupInstance*)

Returns a group instance based on its name

groupName(*groupType*, *groupInstance*)

Form a name for a group instance

joinGroup(*thread*, *groupType*, *groupInstance*, *componentId*, *groupMinSize*)

Operation to create a group instance (with capacity) in a component. Returns the instance

leaveGroup(*group*)

Operation to 'leave' a group by a component. The group will be deactivated / its threads stopped, and deleted..

class riaps.run.dc.**Group**(*parent*, *thread*, *groupType*, *groupInstance*, *componentId*, *groupSpec*, *groupMinSize*)

Bases: object

Group object, represents one group instance that belongs to a component. Acts as the front-end for the GroupThread, channels messages to/from. Some of its methods are run in the GroupThread

GROUP_ACK = b'ack'

GROUP_ANN = b'ann'

GROUP_ERR = b'err'

GROUP_LEL = b'lel'

GROUP_LEX = b'lex'

GROUP_MFL = b'mfl'

GROUP_MJD = b'mjd'

GROUP_MLT = b'mlt'

GROUP_MSG = b'msg'

GROUP_MTL = b'mtl'

GROUP_NLD = b'nld'

GROUP_PARAMETERS = {'consensusTimeout': 1500, 'electionMax': 2000, 'electionMin': 1500, 'heartbeat': 1000, 'peerTimeout': 3000}

GROUP_RCM = b'rcm'

GROUP_RFV = b'rfv'

GROUP_RTC = b'rtc'

GROUP_UPD = b'upd'

getGroupId()

getGroupName()

Group unique name

getLeaderId()

Return the leader's id (or None if no leader)

getSocket()

Returns inproc socket used to communicate with GroupThread

groupSize()

Return the size of the group (>= 1).

groupSocketName(*groupType*, *groupName*, *componentId*)

Forms the unique name for the inproc socket.

handleMessage(*msgFrames=None*)

Receives message from the worker thread and handles it Runs in component thread (called from component polling loop, or from send_port)

hasLeader()

True if the group has a leader.

isLeader()

Return True if the group member IS the leader.

leave()

recv()

Receive a bytes object message from the worker thread Runs in component thread, called from a component

recv_msg(*is_pyobj*)

Receive a message from the worker thread. Message are coming through a message queue. Runs in component thread, called from a component

recv_pyobj()

Receive a Python object message from the worker thread Runs in component thread, called from a component

requestActionVote(*action*, *when*, *kind='consensus'*, *timeout=None*)

requestActionVote_pyobj(*action*, *when*, *kind='consensus'*, *timeout=None*)

requestVote(*topic*, *kind='majority'*, *timeout=None*)

Request a vote on a topic (with timeout). Topic is a bytes. A message is sent to the leader (if any) that starts a voting process. Returns None if there is no leader, otherwise a generated id string for the request.

requestVote_pyobj(*topic*, *kind='majority'*, *timeout=None*)

Request a vote on a topic (with timeout). Topic is a Python object. A message is sent to the leader (if any) that starts a voting process. Returns None if there is no leader, otherwise a generated id string for the request.

send(*msg*)

Sends a bytes object as a message to all members of the group Runs in component thread

sendActionVote(*rfvId*, *vote*)

sendToLeader(*msg*)

Send message to group leader from a member Raise an exception if the group is not coordinated Return False (indicating operation failure) if no leader Runs in component thread.

sendToLeader_pyobj(*msg*)

Send PyObject message to group leader from a member Raise an exception if the group is not coordinated Return False (indicating operation failure) if no leader Runs in component thread.

sendToMember(*msg*, *identity=None*)

Send message to group member (with identity) from the leader If identity is not supplied, last value of identity is used. Raise an exception if the group is not coordinated Return False (indicating operation failure) if no leader Runs in component thread.

sendToMember_pyobj(*msg*, *identity=None*)

Send PyObject message to group member (with identity) from the leader If identity is not supplied, last value of identity is used. Raise an exception if the group is not coordinated Return False (indicating operation failure) if no leader Runs in component thread.

sendVote(*rfvId*, *vote*)

Send a vote (True/False) to the leader on a requested topic identified by the id of the request for vote.

send_port(*msgType*, *msg*, *has_identity=False*)

Send a message to the worker thread. Used by all messages - the messages have multiple frames Runs in component thread

send_pyobj(*msg*)

Sends a Python object as a message to all members of the group Runs in component thread

setup(*groupThread*)

Set up all the sockets for the worker thread Runs in group thread

setupParams()

unsetup(*groupThread*)

Discard all the sockets used in worker thread Runs in group thread

update(*host*, *port*)

Ask the worker thread to update its sockets. Called when the disco responds with the server (pub) host/port pair for the socket(s) to connect to by the client (sub) Runs in component thread

class riaps.run.dc.**GroupThread**(*group*)

Bases: Thread

Worker thread for DC behavior

AUTHORITY = b'ldr'

CANDIDATE = 2

FOLLOWER = 1

HEARTBEAT = b'tic'

LEADER = 3

NO_LEADER = 0

REQVOTE = b'req'

RSPVOTE = b'vot'

announceConsensus(*rfvId*, *vote*)

Announce consensus vote result (yes/no/timeout)

checkAllPolls(*now*)

Check the results of all polls

checkPoll(*poll*, *now*)

Check the result of of one poll

electionTimeout()

Produce a random election timeout

getOwnId()

handleCompMessage()

Handle a message coming from the component

handleMessageForLeader()

Handle message sent to the leader (in leader)

handleMessageForMember()

Handle simple message from leader (in member)

handleNetMessage(*now=None*)

Handle (broadcast) messages coming from the group. Messages could be data messages (to be handed over to the component), peer heartbeat messages, or election-related messages

handleTimeout(*now*)

Handle the timeout on communications within the group

heartbeat(*now*)

Send out a group heartbeat (so that group members can maintain an accurate membership list)

isLeader()

run()

Main loop for GroupThread - polls all sources and calls handlers

sendChangeMessage(*change*, *someId*)

Send message about a change to the component

setLeaderDeadline()

Set deadline for heartbeat from leader

setTimeout(*value*, *now*)

Set the next timeout value, update lastWait (with 'now')

setup()

startPoll(*msg*, *member*)

Start a poll for a member based on message

threshold()

Calculate the threshold for leader election, based on membership list

updatePeers(*now*)

Update membership list (with the current time as the last time a peer was heard from)

updatePoll(*msg*)

Update poll with the vote in msg

updateTimeout(*now*)

Update the timeout value - used when a data message was received from the group; the timeout is updated to reflect elapsed time.

class riaps.run.dc.**Poll**(*parent*, *rfv*, *member*, *timeout*, *deadline*, *numPeers*)

Bases: object

Poll object to represent an active poll in the group leader.

ACTION = 'action'

CONSENSUS = 'consensus'

MAJORITY = 'majority'

VALUE = 'value'

allVoted()

Return True if all peers voted

expired(*now*)

Return True if the voting has expired

result()

Return True if the majority/all voted yes

vote(*vote*)

Count one vote (yes/no)

riaps.run.dcPorts module

Distributed coordination - Communication ports for the groups.

Created on Feb 23, 2019 Author: riaps

class riaps.run.dcPorts.**GroupAnsPort**(*parentPart*, *portName*, *groupSpec*)

Bases: *BindPort*, *GroupDuplexPort*

Group answer port is for the leader to receive messages from members. Based on a DEALER socket. Group-internal communication port for messaging with the leader, no message type, but can be timed.

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

getInfo()

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns

a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type

PortInfo

getPortNumber()

getSocket()

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

get_identity()

inSocket()

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

recv()

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

recvFromMember()

recv_pyobj()

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

reset()

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

send(*_msg*)

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

sendToMember(*msgType*, *msg*)

send_pyobj(*msg*)

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

set_identity(*identity*)

setup()

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

setupSocket(*owner*)

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (*Component*) – The Component the port belongs to. This operation must be called from the component thread only.

update()

Update the socket with information from the discovery service. Abstract, subclasses must implement this method.

Called when the discovery service notifies the actor about a new service provide (e.g. server, publisher, etc.) the port needs to connect to. The operation will perform the connection.

Parameters

- **host** (*str*) – IP address of the service provider
- **port** (*int*) – port number of the service provider

updatePoller(*poller*)

class riaps.run.dcPorts.**GroupDuplexPort**(*parentComponent*, *portName*, *groupSpec*)

Bases: *Port*

class riaps.run.dcPorts.**GroupPubPort**(*parentPart*, *portName*, *groupSpec*)

Bases: *BindPort*, *GroupSimplexPort*

Group Publisher port is for publishing application and housekeeping messages for all group members.

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

getInfo()

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

getSocket()

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

inSocket()

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

recv()

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

recv_pyobj()

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the send_pyobj method.

Returns
a Python data object

Type
any Python data type

reset()

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

send(*msg*)

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

sendGroup(*msgType*, *msg*)

send_pyobj(*msg*)

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

setup()

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

setupSocket(*owner*)

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (Component) – The Component the port belongs to. This operation must be called from the component thread only.

update(*host*, *port*)

Update the socket with information from the discovery service. Abstract, subclasses must implement this method.

Called when the discovery service notifies the actor about a new service provide (e.g. server, publisher, etc.) the port needs to connect to. The operation will perform the connection.

Parameters

- **host** (*str*) – IP address of the service provider
- **port** (*int*) – port number of the service provider

class riaps.run.dcPorts.**GroupQryPort**(*parentPart*, *portName*, *groupSpec*)

Bases: *ConnPort*, *GroupDuplexPort*

Group query port is for accessing the leader from members. Based on a DEALER socket. Group-internal communication port for messaging with the leader, no message type, but can be timed.

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

getInfo()

Retrieve relevant information about this port

getSocket()

Return the socket of port

inSocket()

Return True because the socket is used of input

recv()

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

recvFromLeader()

recv_pyobj()

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

reset()

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

send(*_msg*)

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

sendToLeader(*msgType*, *msg*)

send_pyobj(*msg*)

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

setup()

Set up the port

setupSocket(*owner*)

Set up the socket of the port. Return a tuple suitable for querying the discovery service for the servers (not used currently).

update(*host*, *port*)

Update the query port – connect its socket to a server

class riaps.run.dcPorts.**GroupSimplexPort**(*parentComponent*, *portName*, *groupSpec*)

Bases: *Port*

class riaps.run.dcPorts.**GroupSubPort**(*parentPart*, *portName*, *groupSpec*)

Bases: *ConnPort*, *GroupSimplexPort*

Group subscriber port is for receiving application and housekeeping messages from all group members.

`closeSocket()`

Close down the port. Abstract, subclasses must implement this method.

`getInfo()`

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

`getSocket()`

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

`inSocket()`

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

`recv()`

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

`recvGroup()`

`recv_pyobj()`

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

`reset()`

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

send(*_msg*)

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

send_pyobj(*msg*)

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

setup()

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

setupSocket(*owner*)

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (Component) – The Component the port belongs to. This operation must be called from the component thread only.

riaps.run.deplc module

Deployment manager client Created on Jan 3, 2017

@author: riaps

class riaps.run.deplc.**DeplClient**(*parentActor*, *suffix*)

Bases: object

Deployment service client of an actor

registerActor()

releaseDevice(*bundle*)

reportEvent(*bundle*)

requestDevice(*bundle*)

start()

terminate()

riaps.run.device module

riaps.run.disco module

Created on Oct 19, 2016

@author: riaps

class riaps.run.disco.**DiscoClient**(*parentActor*, *suffix*)

Bases: object

Discovery service client of an actor

handleLookupReq(*bundle*)

handleRegReq(*bundle*)

handleUnlookupReq(*bundle*)

handleUnregReq(*bundle*)

reconnect()

recvFromDisco(*loc*, *shut*)

registerActor()

registerEndpoint(*bundle*)

registerGroup(*bundle*)

rpcDisco(*msgBytes*, *loc*, *shut*)

sendToDisco(*msgBytes*, *loc*, *shut*)

start()

terminate()

unregisterGroup(*bundle*)

riaps.run.dmain module

riaps.run.exc module

Created on Oct 10, 2016

@author: riaps

exception riaps.run.exc.**BuildError**(*message*)

Bases: *RIAPSError*

exception riaps.run.exc.**ControlError**(*message*)

Bases: *RIAPSError*

exception riaps.run.exc.**DatabaseError**(*message*)

Bases: *RIAPSError*

exception riaps.run.exc.**OperationError**(*message*)

Bases: *RIAPSError*

exception riaps.run.exc.**PortError**(*message*, *errno*)

Bases: *RIAPSError*

EAGAIN = 11

EFAULT = 14

EPROTO = 156384763

EROUTE = 113

ETERM = 156384765

exception riaps.run.exc.**RIAPSError**(*message*)

Bases: Exception

exception riaps.run.exc.**SetupError**(*message*)

Bases: *RIAPSError*

exception riaps.run.exc.**StateError**(*message*)

Bases: *RIAPSError*

riaps.run.fsm module

Created on Apr 23, 2020

@author: riaps

class riaps.run.fsm.**FSM**(*initial=None*)

Bases: *Component*

Finite-State Machine component base class.

class **entry**(*state*)

Bases: object

class **exit**(*state*)

Bases: object

fsmLock = <unlocked _thread.RLock object owner=0 count=0>

handleNoTransition(*event*)

handleNondeterminism(*event*, *state*)

handleUnhandledEvent(*event*, *state*)

class **on**(*event*, *state*, *guard=None*, *then=None*)

Bases: object

property state

riaps.run.insPort module

Created on Jan 9, 2017

@author: riaps

class riaps.run.insPort.**InsPort**(*parentPart*, *portName*, *portSpec*)

Bases: *Port*

classdocs

activate()

Activate the port object. Subclasses can override this method.

deactivate()

Deactivate the port object. Subclasses can override this method.

getContext()

getInfo()

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

getSocket()

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

get_identity()

get_plug_identity(*plug*)

inSocket()

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

ins_port_recv(*is_pyobj*)

ins_port_send(*msg*, *is_pyobj*)

recv()

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

`recv_pyobj()`

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

`reset()`

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

`send(msg)`

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

`send_pyobj(msg)`

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

`set_identity(identity)`

`setup()`

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

`setupPlug(thread)`

`setupSocket(owner)`

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (`Component`) – The Component the port belongs to. This operation must be called from the component thread only.

`terminate()`

Terminate all activities of the port. Subclasses can override this method.

riaps.run.main module

riaps.run.part module

Part class Created on Oct 9, 2016

@author: run

class riaps.run.part.**Part**(*parentActor*, *iTypeDef*, *iName*, *iTypeName*, *iArgs*)

Bases: object

Part class to encapsulate and manage component (and its thread)

class State(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Active = 3

Checkpointing = 4

Destroyed = 7

Inactive = 5

Initial = 1

Passive = 6

Ready = 2

Starting = 0

activate()

Activate this part

buildAllPorts(*portSpecs*)

Build all the ports of the part

buildPorts(*res*, *key*, *ports*, *class_*)

Build the port objects of a kind of this part

checkpoint()

deactivate()

destroy()

getActorID()

getActorName()

getAppName()

getName()

getTypeName()

getUUID()

handleCPULimit()

handleMemLimit()

handleNICStateChange(*state*)

handleNetLimit()

handlePeerStateChange(*state*, *uuid*)

handlePortUpdate(*portName*, *host*, *port*)
Handle a port update message coming from the discovery service

handleReinstate()
Reinstate providers with a restarted disco

handleSpcLimit()

load()
Load the component implementation code

property mods

passivate()

reactivate()

sendControl(*cmd*, *timeOut*)
Send a control message to component thread

setup(*control_*)
Set up the part and change its state to Ready

setupPorts(*ports*)
Set up all the ports of this part

terminate()

riaps.run.peripheral module

Peripheral class - encapsulates a device, used in an app actor Created on Jan 6, 2017

@author: riaps

class riaps.run.peripheral.**Peripheral**(*parentActor*, *iTypeDef*, *iName*, *iTypeName*, *iArgs*)
Bases: object

Peripheral class to encapsulate access to a device component Note: implements a public interface compatible with a part

class State(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)
Bases: Enum

Active = 3

Checkpointing = 4

Destroyed = 7

Inactive = 5

Initial = 1

Passive = 6

Ready = 2

Starting = 0

activate()
Activate this peripheral

checkpoint()

deactivate()

destroy()

getControl()

handleCPULimit()

handleMemLimit()

handleNICStateChange(*state*)

handleNetLimit()

handlePeerStateChange(*state*, *uuid*)

handlePortUpdate(*_portName*, *_host*, *_port*)
Handle an update message coming from the devm service

handleReinstate()

handleSpcLimit()

property mods

passivate()

reactivate()

setup()
Set up the peripheral and change its state to Ready

terminate()

riaps.run.port module

Base class for all Port objects

class riaps.run.port.**BindPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *Port*

closeBindSocket()

setupBindSocket(*owner*, *zmqType*, *portKind*, *sockopts=[]*)

Set up a bind socket

class riaps.run.port.**ConnPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *Port*

closeConnSocket()

connected()

Return the number of servers this port is connected to.

resetConnSocket(*zmqType*, *sockopts=[]*)

Reset a conn socket: remove and recreate

setupConnSocket(*owner*, *zmqType*, *portKind*, *sockopts=[]*)

Setup a conn socket

update(*host*, *port*)

Update the client – connect its socket to a server

class riaps.run.port.**DuplexBindPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *BindPort*, *DuplexPort*

class riaps.run.port.**DuplexConnPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *ConnPort*, *DuplexPort*

class riaps.run.port.**DuplexPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *Port*

class riaps.run.port.**Port**(*parentPart*, *portName*, *portSpec=None*)

Bases: object

Base class for all Port objects.

Port objects are used by a component to communicate with other components, in the same process, on the same host, or on the same network. Ports encapsulate low-level communication objects (zeromq sockets).

Parameters

- **parentPart** (*Part*) – the Part object that owns this port.
- **portName** (*str*) – the name of the port (from the model)

activate()

Activate the port object. Subclasses can override this method.

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

deactivate()

Deactivate the port object. Subclasses can override this method.

`getDeadline()`

Return the deadline parameter associated with the port's operation

Returns
Deadline for the operation associated with the port, in seconds.

Return type
float

`getGlobalIface()`

Return the IP address of the global network interface

The operation retrieves the result from the parent actor and caches it.

Returns
Global IP address of the form xxx.xxx.xxx.xxx

Return type
str

`getIndex()`

Return the index of the port.

For input ports the index is a small integer indicating its position in the port list of the component, for non-input ports it is None. The index is used to determine the priority order for the port among all the ports, the concrete value is irrelevant.

Returns
Index value for the port among all input ports.

Return type
int

`getInfo()`

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

`getLocalIface()`

Return the IP address of the local network interface (typically 127.0.0.1)

The operation retrieves the result from the parent actor and caches it.

Returns
Local IP address of the form xxx.xxx.xxx.xxx

Return type
str

`getSocket()`

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

get_hwm()

Retrieve the high-water mark for the socket.

get_recvTime()

Return the timestamp taken at the last receive operation.

Returns
time of the last message receive operation

Return type
float

get_recv_timeout()

Retrieve the receive timeout parameter for the port.

Receive timeout determines how long a receive operation will block before throwing a PortError.EAGAIN exception. None means infinite timeout.

Returns
None (if no timeout is set) or the timeout value in seconds.

Return type
None or float

get_sendTime()

Return the timestamp of the sending time of the last message receive.

Returns
time when the last message received was sent

Return type
float

get_send_timeout()

Retrieve the send timeout parameter for the port.

Send timeout determines how long a send operation will block before throwing a PortError.EAGAIN exception. None means infinite timeout.

Returns
None (if no timeout is set) or the timeout value in seconds.

Return type
None or float

inSocket()

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

port_recv(*is_pyobj*)

Lowest level message receiving operation. Subclasses can override this operation.

The message is received as a multi-part message.If the receiving port is timed, the current timestamp is saved as the time of message reception. If the message is to be received as a Python object, it is unpickled, otherwise the message is returned as is (as a bytes). If the message included a timestamp, it is retrieved and saved as the time of message sending.

Parameters
is_pyobj (*bool*) – flag to indicate if the expected message is a Python data object.

Returns
the message received

Except
Throws a PortError exception when the underlying network operation fails.

port_send(*msg*, *is_pyobj*)

Lowest level message sending operation. Subclasses can override this operation.

If the message is to be sent as a Python object, it is pickled; otherwise it is assumed to be a bytes. The message is packed into a frame, and, if the port is 'timed' a current timestamp is appended as another frame. The message is sent as a multi-part message.

Parameters

- **msg** (*either a bytes or any Python data object*) – message to be sent
- **is_pyobj** (*bool*) – flag to indicate if the message is a Python object.

Returns
True

Except
Throws a PortError exception when the underlying network operation fails.

recv()

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

recv_capnp()

DEPRECATED. Receive an object (if possible) through the port

recv_pyobj()

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the send_pyobj method.

Returns
a Python data object

Type
any Python data type

reset()

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

send(*msg*)

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

send_capnp(*msg*)

DEPRECATED. Send a byte array (if possible) out through the port

send_pyobj(*msg*)

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

setOwner(*owner*)

Save owner thread into a data member.

Parameters
owner (ComponentThread) – The ComponentThread the port is handled in.

set_recv_timeout(*rto*)

Set the receive timeout for the port.

Receive timeout determines how long a receive operation will block before throwing a PortError.EAGAIN exception. None means infinite timeout.

Parameters
rto (*None or float*) – None (if no timeout is set) or the timeout value in seconds.

set_send_timeout(*sto*)

Set the send timeout for the port.

Send timeout determines how long a send operation will block before throwing a PortError.EAGAIN exception. None means infinite timeout.

Parameters
sto (*None or float*) – None (if no timeout is set) or the timeout value in seconds.

set_sockoptions(*sockopts*)

setup()

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

setupCurve(*server*)

setupSocket(*owner*)

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (Component) – The Component the port belongs to. This operation must be called from the component thread only.

terminate()

Terminate all activities of the port. Subclasses can override this method.

update(*host*, *port*)

Update the socket with information from the discovery service. Abstract, subclasses must implement this method.

Called when the discovery service notifies the actor about a new service provide (e.g. server, publisher, etc.) the port needs to connect to. The operation will perform the connection.

Parameters

- **host** (*str*) – IP address of the service provider
- **port** (*int*) – port number of the service provider

class riaps.run.port.**PortInfo**(*portKind*, *portScope*, *portName*, *msgType*, *portHost*, *portNum*)

Bases: tuple

msgType

Alias for field number 3

portHost

Alias for field number 4

portKind

Alias for field number 0

portName

Alias for field number 2

portNum

Alias for field number 5

portScope

Alias for field number 1

class riaps.run.port.**PortScope**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

GLOBAL = 1

INTERNAL = 3

LOCAL = 2

scope()

class riaps.run.port.**SimplexBindPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *BindPort*, *SimplexPort*

class riaps.run.port.**SimplexConnPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *ConnPort*, *SimplexPort*

class riaps.run.port.**SimplexPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *Port*

riaps.run.pubPort module

Created on Oct 10, 2016

@author: riaps

class riaps.run.pubPort.**PubPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *SimplexBindPort*

Publisher port

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

getInfo()

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

getSocket()

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

inSocket()

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

recv()

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

recv_pyobj()

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

`reset()`

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

`send(msg)`

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

`send_pyobj(msg)`

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

`setup()`

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

`setupSocket(owner)`

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (Component) – The Component the port belongs to. This operation must be called from the component thread only.

`update(host, port)`

Update the socket with information from the discovery service. Abstract, subclasses must implement this method.

Called when the discovery service notifies the actor about a new service provide (e.g. server, publisher, etc.) the port needs to connect to. The operation will perform the connection.

Parameters

- **host** (*str*) – IP address of the service provider
- **port** (*int*) – port number of the service provider

riaps.run.qryPort module

Query port class Created on Oct 10, 2016

@author: riaps

class riaps.run.qryPort.**QryPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *DuplexConnPort*

Query port is to access a server. Has a request and a response message type, and uses a DEALER socket.

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

getInfo()

Retrieve relevant information about this port

getSocket()

Return the socket of port

inSocket()

Return True because the socket is used of input

recv()

Receive a bytes through this port

recv_pyobj()

Receive an object through this port

reset()

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

send(*msg*)

Send bytes through this port

send_pyobj(*msg*)

Send an object through this port

setup()

Set up the port

setupSocket(*owner*)

Set up the socket of the port. Return a tuple suitable for querying the discovery service for the publishers

riaps.run.repPort module

Created on Oct 10, 2016

@author: riaps

class riaps.run.repPort.**RepPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *DuplexBindPort*

Similar to a server port.

`closeSocket()`

Close down the port. Abstract, subclasses must implement this method.

`getInfo()`

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

`getSocket()`

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

`inSocket()`

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

`recv()`

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

`recv_pyobj()`

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

`reset()`

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

send(*msg*)

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

send_pyobj(*msg*)

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

setup()

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

setupSocket(*owner*)

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (Component) – The Component the port belongs to. This operation must be called from the component thread only.

update(*host*, *port*)

Update the socket with information from the discovery service. Abstract, subclasses must implement this method.

Called when the discovery service notifies the actor about a new service provide (e.g. server, publisher, etc.) the port needs to connect to. The operation will perform the connection.

Parameters

- **host** (*str*) – IP address of the service provider
- **port** (*int*) – port number of the service provider

riaps.run.reqPort module

Created on Oct 10, 2016

@author: riaps

class riaps.run.reqPort.**ReqPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *DuplexConnPort*

Similar to a client port

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

`getInfo()`

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

`getSocket()`

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

`inSocket()`

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

`recv()`

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

`recv_pyobj()`

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

`reset()`

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

`send(msg)`

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

`send_pyobj`(*msg*)

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

`setup`()

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

`setupSocket`(*owner*)

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (Component) – The Component the port belongs to. This operation must be called from the component thread only.

riaps.run.srvPort module

Created on Oct 10, 2016

@author: riaps

`class riaps.run.srvPort.SrvPort`(*parentComponent*, *portName*, *portSpec*)

Bases: *DuplexBindPort*

classdocs

`closeSocket`()

Close down the port. Abstract, subclasses must implement this method.

`getInfo`()

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

`getSocket`()

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

`inSocket()`

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

`recv()`

Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

`recv_pyobj()`

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

`reset()`

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

`send(msg)`

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

`send_pyobj(msg)`

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

`setup()`

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

setupSocket(*owner*)

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (Component) – The Component the port belongs to. This operation must be called from the component thread only.

update(*host*, *port*)

Update the socket with information from the discovery service. Abstract, subclasses must implement this method.

Called when the discovery service notifies the actor about a new service provide (e.g. server, publisher, etc.) the port needs to connect to. The operation will perform the connection.

Parameters

- **host** (*str*) – IP address of the service provider
- **port** (*int*) – port number of the service provider

riaps.run.subPort module

Created on Oct 10, 2016

@author: riaps

class riaps.run.subPort.**SubPort**(*parentComponent*, *portName*, *portSpec*)

Bases: *SimplexConnPort*

Subscriber port

closeSocket()

Close down the port. Abstract, subclasses must implement this method.

getInfo()

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

getSocket()

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

inSocket()

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

`recv()`
Receive a byte array (if possible) through the port

Used for receiving a message that is subsequently deserialized.

Returns
a message packed into a bytes.

Return type
bytes

`recv_pyobj()`
Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

`reset()`
Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

`send(_msg)`
Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

`send_pyobj(msg)`
Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

`setup()`
Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

`setupSocket(owner)`
Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (Component) – The Component the port belongs to. This operation must be called from the component thread only.

riaps.run.timPort module

Created on Oct 10, 2016

@author: riaps

class riaps.run.timPort.**TimPort**(*parentPart*, *portName*, *portSpec*)

Bases: *Port*

Timer port

activate()

Activate the timer port

cancel()

Cancel the sporadic timer

deactivate()

Deactivate the timer port

getDelay()

Get the current delay (for sporadic timer)

getInfo()

Retrieve configuration information about the port. Abstract, subclasses must implement this method.

Returns
a tuple containing the name of the port's type: req,rep,clt,srv,qry,ans,pub,sub,ins,or tim; the kind of the port (global, local, internal); the name of the port object; the name of the message type; the host and the port number.

Return type
PortInfo

getPeriod()

Read the period of the periodic timer

getSocket()

Return the socket(s) used by the port object. Abstract, subclasses must implement this method.

Returns
a low-level socket

Return type
zmq.Socket

halt()

Halt the timer

inSocket()

Return True if the socket can be used for input. Abstract, subclasses must implement this method.

Returns
logical value indicating whether the socket is for input.

Type
bool

launch()

Launch (start) the sporadic timer

recv()

Receive time stamp (a float) as a byte array

recv_pyobj()

Receive a Python data object (if possible) through the port. Abstract, subclasses must implement this method.

The raw message received is deserialized using pickle and returned. Messages received this way had to be sent using the `send_pyobj` method.

Returns
a Python data object

Type
any Python data type

reset()

Reset the port object. Subclasses can override this method.

Reset is to be used when a send or receive operation fails and the port needs to be re-initialized.

running()

Returns True if the timer is running

send()

Send a byte array (if possible) out through the port.

Used for sending a message that has been serialized into bytes previously.

Parameters
msg – the message packed into a bytes

Return type
bytes

send_pyobj(*msg*)

Send a Python data object (if possible) out through the port. Abstract, subclasses must implement this method.

The object is serialized using pickle and sent. Messages sent using this method are received using the `recv_pyobj` method.

Parameters
msg (*any Python data type*) – the message to be sent.

setDelay(*_delay*)

Set the current delay (for sporadic timer)

setPeriod(*_period*)

Set the period - will be changed after the next firing. Period must be positive

setup()

Initialize the port object (after construction but before socket creation). Abstract, subclasses must implement this method.

setupSocket(*owner*)

Setup the socket. Abstract, subclasses must implement this method.

Parameters
owner (Component) – The Component the port belongs to. This operation must be called from the component thread only.

terminate()

Terminate the timer

class riaps.run.timPort.TimerThread(*parent*)

Bases: Thread

class Command(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

ACTIVATE = 2

CANCEL = 5

DEACTIVATE = 3

HALT = 6

START = 4

TERMINATE = 1

cmdError(*where*, *cmd*)

getDelay()

Get the current delay (for sporadic timer)

getPeriod()

Read out the period

ready()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

running()

Returns True if the timer is running

setDelay(*_delay*)

Set the current delay (for sporadic timer)

setPeriod(*_period*)

Set the period (for periodic timer). Takes effect after the next firing.

waitFor(*timeout=None*)

Module contents

1.1.11 riaps.utils package

Submodules

riaps.utils.appdesc module

Application descriptor (used as a yaml object)

Created on Oct 29, 2018

@author: riaps

class riaps.utils.appdesc.**AppDescriptor**(*url*, *host*, *mac*, *sha*, *home*, *hosts*, *network*)

Bases: YAMLObject

RIAPS app origin - 'signature file url = URL of the repo (or local folder) the app is coming from host = host IP addares mac = MAC address of host sha = SHA of package home = local source folder (used in remote debugging) hosts = hosts participating in the app network = network access control for nodes, (ip|'[]') => [] | [('dns' | ip)]+

yaml_loader

alias of SafeLoader

riaps.utils.config module

Created on Nov 23, 2016

@author: riaps

class riaps.utils.config.**Config**

Bases: object

Configuration database for RIAPS tools Including logging configuration

ACTOR_DEBUG_SERVER = ''

APP_LOGS = ''

CTRL_DEBUG_SERVER = ''

CTRL_HEARTBEAT = True

DEPLO_DEBUG_SERVER = ''

DEVICE_DEBUG_SERVER = ''

DISCO_DEBUG_SERVER = ''

DISCO_TYPE = 'redis'

NETMON = True

NIC_CEIL = '131kbps'

NIC_NAME = None

NIC_RATE = '118kbps'

```
NODE_HEARTBEAT = True

RECV_HWM = 100

RECV_TIMEOUT = -1

SECURITY = True

SEND_HWM = 100

SEND_TIMEOUT = -1

TARGET_USER = 'riaps'
```

riaps.utils.ctrlhost module

riaps.utils.gencert module

Script to generate a public/private key pair and a self-signed certificate for securing riaps communications. THE KEY ANND CERTIFICATE MUST NEVER BE USED IN FIELDED SYSTEMS. Keys and the certificate must be installed in the $RIAPSHOME/keys directory. Private key is NOT ENCRYPTED.

@author: riaps

riaps.utils.gencert.**generate_keys**(*cert_dir*)

riaps.utils.gencert.**generate_self_signed_cert**(*cert_dir*, *key*)

riaps.utils.gencert.**generate_zmq_cert**(*cert_dir*)

riaps.utils.gencert.**main**()

riaps.utils.ifaces module

Various network interface utility functions Created on Nov 4, 2016

@author: riaps

riaps.utils.ifaces.**getNetworkInterfaces**(*nicName=None*)

Determine the IP address of the network interfaces Return a tuple of list of global IP addresses, list of MAC addresses, and local IP address If the requested interface is found the list will contain the information for that interface only.

riaps.utils.ifaces.**get_random_port**()

Get a random open port

riaps.utils.ifaces.**get_unix_dns_ips**()

Retrieve the IP address(es) of dns servers used by this host

riaps.utils.ifaces.**is_valid_ipv4_address**(*address*)

Determine if the argument is a valid IP address

riaps.utils.names module

name operations Created on Jan 19, 2018

@author: riaps

riaps.utils.names.**actorIdentity**(*appName*, *actorName*, *pid*)

riaps.utils.singleton module

Created on Jul 23, 2018

@author: riaps

riaps.utils.singleton.**singleton**(*process_name*, *suffix=None*)

Enforce the caller process is a singleton

riaps.utils.spdlog_setup module

Created on Jan 3, 2019

@author: riaps

riaps.utils.spdlog_setup.**add_basic_file_sink_mt**(*s*)

riaps.utils.spdlog_setup.**add_basic_file_sink_st**(*s*)

riaps.utils.spdlog_setup.**add_color_stdout_sink_mt**(*_s*)

riaps.utils.spdlog_setup.**add_color_stdout_sink_st**(*_s*)

riaps.utils.spdlog_setup.**add_daily_file_sink_mt**(*s*)

riaps.utils.spdlog_setup.**add_daily_file_sink_st**(*s*)

riaps.utils.spdlog_setup.**add_null_sink_mt**(*_s*)

riaps.utils.spdlog_setup.**add_null_sink_st**(*_s*)

riaps.utils.spdlog_setup.**add_parent_dir**(*s*)

riaps.utils.spdlog_setup.**add_rotating_file_sink_mt**(*s*)

riaps.utils.spdlog_setup.**add_rotating_file_sink_st**(*s*)

riaps.utils.spdlog_setup.**add_stdout_sink_mt**(*_s*)

riaps.utils.spdlog_setup.**add_stdout_sink_st**(*_s*)

riaps.utils.spdlog_setup.**add_syslog_sink_mt**(*s*)

riaps.utils.spdlog_setup.**add_syslog_sink_st**(*s*)

riaps.utils.spdlog_setup.**add_tcp_sink_mt**(*s*)

riaps.utils.spdlog_setup.**add_tcp_sink_st**(*s*)

riaps.utils.spdlog_setup.**file_size**(*num*)

riaps.utils.spdlog_setup.**from_file**(*fname*)

riaps.utils.spdlog_setup.**get_logger**(*name*)

riaps.utils.sudo module

sudo operations Created on Jan 19, 2018

@author: riaps

riaps.utils.sudo.**is_su**()

riaps.utils.sudo.**riaps_sudo**(*cmd*, *timeout=None*)

riaps.utils.ticker module

Created on Jun 21, 2022

@author: riaps

class riaps.utils.ticker.**Ticker**(*period*, *callback*)

Bases: Thread

Simple periodic ticker

cancel()

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

riaps.utils.trace module

Created on Dec 16, 2017

@author: riaps

riaps.utils.trace.**riaps_trace**(*debug=None*, *prog=None*)

Setup trace mode and wait for the debug server.

@debug: Debug server control string of the form 'hostname:portname',
both of which are optional, ':' defaulting to localhost:5678

@prog: Label for the program, looked up in the riaps configuration file.

First, it attempts to connect to the debug server using the debug argument (if present). Second, it tries to connect to the debug server using the information from the configuration file. If the config file argument is empty, it silently returns. Returns: True of False depending on whether the program is running in trace mode.

riaps.utils.trace.**riaps_trace_setup**(*debug*)

Module contents

1.2 Submodules

1.2.1 riaps.riaps_actor module

1.2.2 riaps.riaps_ctrl module

1.2.3 riaps.riaps_depll module

Top-level script of the deployment language processor

Example:

```
riaps_depll model [-v | --verbose] [-g|--generate]
```

Arguments:

- `model`: name of model file to be processed
- `-v|--verbose`: print the resulting JSON file on the console
- `-g`: generate a JSON file

1.2.4 riaps.riaps_deplo module

1.2.5 riaps.riaps_device module

1.2.6 riaps.riaps_disco module

1.2.7 riaps.riaps_fab module

Top level script to start fabric file for handling multiple RIAPS nodes setup

Created on March 6, 2019

Arguments:

- `fabcmd`: fabric command desired

optional arguments: - `-H | --hosts hostnames`: list of hostnames (comma separated) - `-R | --roles rolenames`: list of roles (comma separated) - `-f hostfilename` : absolute path to local host file - `-i ssh private key` : relative or absolute path to specific private key

If specific hostnames are not given, the command will be called for all hosts listed in /usr/local/riaps/etc/riaps_hosts.conf

@author: riaps

`riaps.riaps_fab.`**`bash`**`(`*`cmd`*`)`

1.2.8 riaps.riaps_gen module

Top-level script to start the language processor ('lang') for app models Created on Nov 15, 2018

Arguments:

-m, –model : Full path of the model.json. -o, –output : Output directory. Default is the directory of the model file. -cpp, –cpp : List of components to be generated in C++. -py, –python : List of components to be generated in Python. -s, –ser : Message serializer ('capnp' or 'pickle'). -w, –overwrite : Overwrite existing code (no sync).

@author: riaps

1.2.9 riaps.riaps_gviz module

Top-level script to start the graphic visualization processor ('gviz')

Example:

```
riaps_gviz model deplo [-v|--verbose]
```

The script generates a .dot file shown the allocation of components and actors to target nodes based on the model and deployment files.

Arguments:

`-model`: name of application model file to be processed - `deplo`: name of deployment model file to be processed - `-v|--verbose`: prints the JSON produced from the deployment model

Output:

- `appname.dot` : graphviz-style dot file for the name application (based on the model)

1.2.10 riaps.riaps_lang module

Top-level script to start the language processor ('lang') for app models

Example:

```
riaps_lang model [-v|--verbose]
```

The program analyzes the model file and generates a JSON file.

Arguments:

- `model` : Name of model file to be processed
- `-v|--verbose`: print the resulting JSON file on the console

1.2.11 riaps.riaps_logger module

1.2.12 riaps.riaps_mn module

1.3 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

A

D

E

F

G

H

I

J

L

M

N

S

T

U

V

W

Y