aiomas Documentation

Release 0.5.0

Stefan Scherfke

July 06, 2015

Contents

PyPI | Bitbucket | Mailing list | IRC: #aiomas

aiomas is an easy-to-use library for *remote procedure calls (RPC)* and *multi-agent systems (MAS)*. It's written in pure Python on top of asyncio.

Here is an example how you can write a simple multi-agent system:

```
>>> import asyncio
>>> import aiomas
>>>
>>> class TestAgent(aiomas.Agent):
...     def __init__(self, container, addr):
...         super().__init__(container, addr)
...         print('Ohai, I am %s' % self)
...
...     @asyncio.coroutine
...     def run(self, addr):
...         remote_agent = yield from self.container.connect(addr)
...         ret = yield from remote_agent.service(42)
...         print('%s got %s from %s' % (self, ret, remote_agent))
...
...     @aiomas.expose
...     def service(self, value):
...         return value
>>>
>>> c = aiomas.Container(('localhost', 5555))
>>> agents = [c.spawn(TestAgent) for i in range(2)]
Ohai, I am TestAgent('tcp://localhost:5555/0')
Ohai, I am TestAgent('tcp://localhost:5555/1')
>>> aiomas.run(until=agents[0].run(agents[1].addr))
TestAgent('tcp://localhost:5555/0') got 42 from TestAgentProxy('tcp://localhost:5555/1')
>>> c.shutdown()
```

aiomas is released under the MIT license. It requires Python 3.4 and above and runs on Linux, OS X, and Windows.

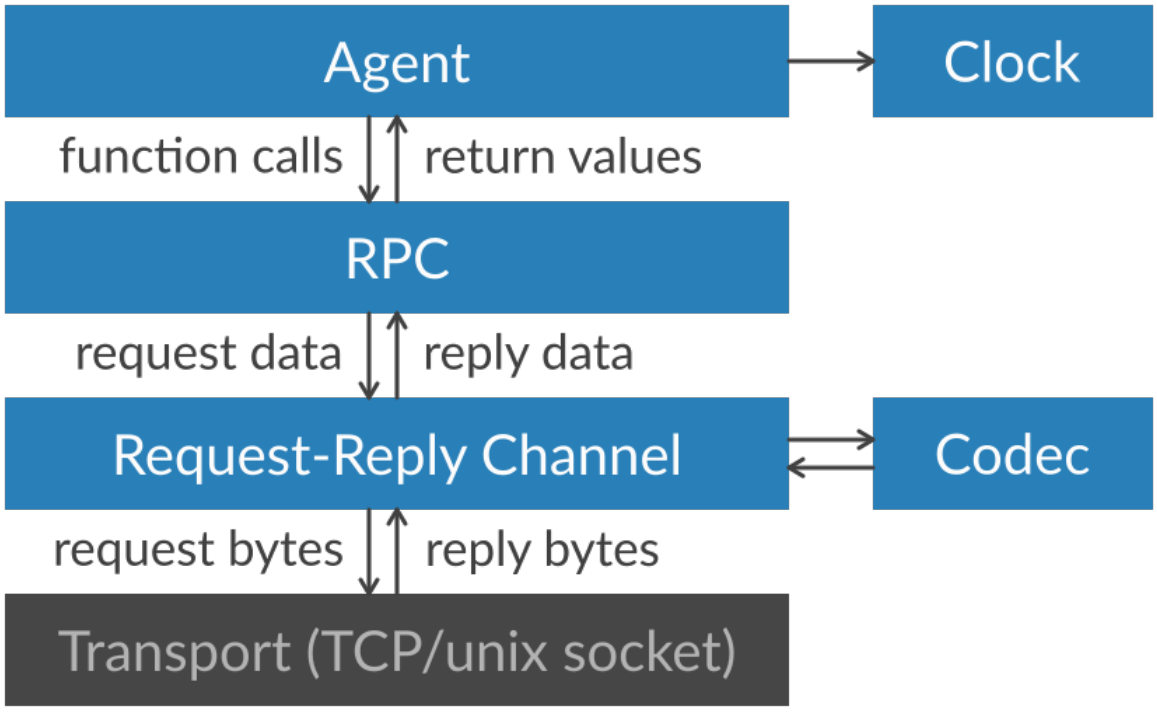
CHAPTER 1

Contents:

1.1 Overview

Aiomas' main goal is making it easier to create distributed systems (like multi-agent systems (MAS)) with pure Python and asyncio.

Therefore, it adds three layers of abstraction around the transports (TCP or Unix domain sockets) that asyncio provides:



1. The channel layer allows you to send and receive actual data like strings, lists ob numbers instead of single bytes.

   The `Channel` class lets you make *requests* and wait for the corresponding *replies* within a coroutine: `reply = yield from channel.send(request)`.

   Every *channel* has a `Codec` instance that is responsible for (de)serializing the data that is being sent via the channel. By default, JSON is used for that. Alternatively, you can use MsgPack and optionally compress it using Blosc. You can also extend codecs with custom serializers for more object types.

2. The remote procedure call (RPC) layer lets you call function on remote objects.

   You can expose the methods of an object as well as functions within a dict. On the other side of the connection, proxy objects represent these exposed functions.

   You can call remote functions within a coroutine: `return_value = yield from remote.method('spam', eggs=3.14)`.

3. The agent layer hides some of the *RPC* layer's complexity and allows you to create thousands of interconnected objects *(agents)* without opening thousands of unique connections between them.

   Therefore, all agents live within a *container*. Containers take care of creating agent instances and performing the communication between them.

   The container provides a *clock* for the agents. This clock can either be synchronized with the real (wall-clock) time or be set by an external process (e.g., other simulators).

The following sections explain theses layers in more detail.

1.2 The agent layer

This section describes the agent layer and gives you enough information to implement your own multi-agent system without going too much into detail. For that, you should also read the section about the RPC layer.

1.2.1 Overview

You can think of agents as small, independent programs running in parallel. Each agent waits for input (e.g., incoming network messages), processes the input and creates, based on its internal state and the input, some output (like outgoing network messages).

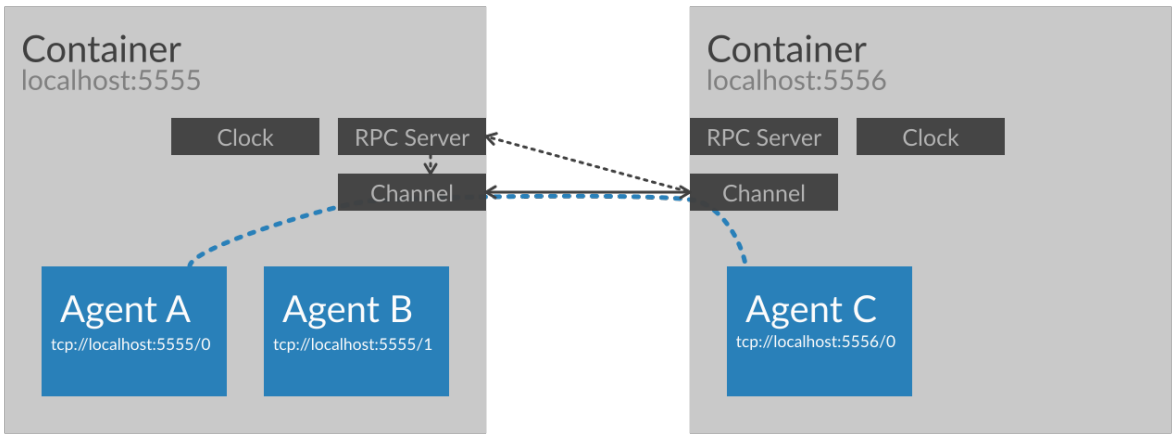
You can also imagine them as being like normal objects that call other object's methods. But instead of calling these methods directly, they do remote procedure calls (RPC) via a network connection.

In theory, that means that every agent has a little server with an event loop that waits for incoming messages and dispatches them to the corresponding method calls.

Using this model, you would quickly run out of resources with hundreds or thousands of interconnected agents. For this reason, agents are clustered in containers. A container provide the network server and event loop which all agents within the container share.

Agents are uniquely identified by the container's address and an ID (which is unique within a container), for example: *tcp://localhost:5555/0*.

The following image illustrates this: If *Agent C* wants to send a message to *Agent A*, its container connects to *A's* container. *Agent C* can now send a message to *Agent A*. If *Agent C* now wanted to send a message to *Agent B*, it would simply reuse the same connection.



As you can see in the figure above, containers also have a *clock*, but you can ignore that fact for the moment. We'll come back to that later.

So the four components of a distributed system in aiomas are:

1. **Agent:** You implement your business logic in subclasses of `aiomas.Agent`. Agents can be *reactive* or *proactive*.

   Reactive agents only react to incoming messages, that means, they simply expose some methods that other agents can call.

   Proactive agents actively perform one ore more tasks, i.e., calling other agent's methods.

   An agent can be both, *proactive* and *reactive*.

2. **Container:** All agents live in a container. The agent container implements everything networking related (e.g., a shared RPC server) so that the agent base class can be as light-weight as possible. It also defines the *codec* used for message (de)serialization and provides a *clock* for agents.

3. **Codec:** Codecs define how messages to other agents get serialized to byte strings that can be sent over the network. The base codecs can only serialize the most common object types (like numbers, strings, lists or dicts) but you can extend them with serializers for custom object types.

   The Codecs section explain all this in detail.

4. **Clock:** Every container provides a clock for agents. Clocks are important for operations with a timeout (like `sleep()`). The default clock is a real-time clock synchronized to your system's time.

   However, if you want to integrate your MAS with a simulation, you may want to let the time pass faster then real-time (in order to decrease the duration of your simulation). For that use case, aiomas provides a clock that can be synchronized with external sources.

   All clocks provide functions to get the current time, sleep for some time or execute a task after a given timeout. If you use these function instead of the once asyncio provides, you can easily switch between different kinds of clocks. The Clocks section provides more details and examples.

Don't worry if you feel a bit confused now. I'll explore all of this with small, intuitive examples.

1.2.2 Hello World: A single, proactive agent

In our first example, we'll create a very simple agent which repeatedly prints "Hello, World!":

```
>>> from asyncio import coroutine
>>> import aiomas
>>>
>>> class HelloWorld(aiomas.Agent):
...     def __init__(self, container, addr, name):
...         super().__init__(container, addr)
...         self.name = name
...
...     @coroutine
...     def run(self):
...         print(self.name, 'says:')
...         clock = self.container.clock
...         for i in range(3):
...             yield from clock.sleep(0.1)
...             print('Hello, World!')
```

Agents should be a subclass of *Agent*. They'll receive a reference to their container and their address which they must pass to their super class.

Our agent also defines a task `run()` which prints "Hello, World!" three times. The task also uses the container's clock to sleep for a small amout of time between each print.

The clock (see *clocks*) exposes various time related functions similar to those that asyncio offers, but you can easily exchange the default real-time clock of a container with another one (e.g., one where time passes faster than real-time, which is very useful in simulations).

```
>>> container = aiomas.Container(('localhost', 5555))
>>> agent = container.spawn(HelloWorld, 'Monty')
>>> aiomas.run(until=agent.run())
Monty says:
Hello, World!
Hello, World!
Hello, World!
>>> container.shutdown()
```

In order to run the agent, you need to start a *Container* first. The container will create an RPC server and bind it to the specified address.

You don't instantiate your agent directly but via the *Container.spawn()* method, which will also register the agent with the container. The method receives the agent class and, optionally, (keyword) arguments for it. These arguments are passed in addition to the container and address to the agent's `__init__()`.

The function *run()* is an alias for `loop = asyncio.get_event_loop(); loop.run_until_complete(task)`.

These are the very basics auf aiomas' agent module. In the next section you'll learn how an agent can call another agent's methods.

1.2.3 Calling other agent's methods

The purpose of multi-agent systems is having multiple agents calling each other's methods. Let's see how we do this. For the sake of simplicity we'll create two different agent types in this example where `Caller` calls a method of `Callee`:

```
>>> import asyncio
>>> import aiomas
>>>
>>> class Callee(aiomas.Agent):
...
...     @aiomas.expose
...     def spam(self, times):
...         """Return a lot of spam."""
...         return 'spam' * times
>>>
>>>
>>> class Caller(aiomas.Agent):
...
...     @asyncio.coroutine
...     def run(self, callee_addr):
...         print(self, 'connecting to', callee_addr)
...         callee = yield from self.container.connect(callee_addr)
...         print(self, 'connected to', callee)
...         result = yield from callee.spam(3)
...         print(self, 'got', result)
>>>
>>>
>>> container = aiomas.Container(('localhost', 5555))
>>> callee = container.spawn(Callee)
>>> caller = container.spawn(Caller)
>>> aiomas.run(until=caller.run(callee.addr))
Caller('tcp://localhost:5555/1') connecting to tcp://localhost:5555/0
Caller('tcp://localhost:5555/1') connected to CalleeProxy('tcp://localhost:5555/0')
Caller('tcp://localhost:5555/1') got spamspamspam
>>> container.shutdown()
```

The agent `Callee` exposes its method `spam()` via the `@aiomas.expose` decorator and thus allows other agents to call this method. The arguments and return values of exposed methods need to be serializable (the next sections shows you how to add serializers for custom data types). Furthermore, exposed methods can be both, normal functions and coroutines.

The `Caller` agent does not expose any methods, but defines a task `run()` which receives the address of the remote agent. It can connect to that agent via the container's *`connect()`* method. This is a coroutine, so you need to `yield from` it. It's return value is a proxy object to the remote agent.

Proxies represent a remote object and provide access to exposed attributes (like functions) of that object. In the example above, we use the proxy to call the `spam()` function. Since this involves sending messages to the remote agent, you always need to use `yield from` with remote method calls.

- Many agents one container
- Many agents multiple containers on one machine
- many agents, multiple machines.

1.3 The RPC layer

1.4 The channel layer

1.5 Codecs for message serialization

Codecs are used to convert the objects that you are going to send over the network to bytes and the bytes that you received back to the original objects. This is called *serialization* and *deserialization*.

A codec specifies, how the text representation of a certain object looks like. It can also recreate the object based on its text representation.

For example, the JSON encoded representation of the list `['spam', 3.14]` would be `b'["spam",3.14]'`.

Many different codecs exists. Some of the most widely used ones are JSON, XML or MsgPack. They mainly differ in their:

- verbosity or compactness: How many bytes are needed to encode an object?
- performance: How fast can they encode and decode objects?
- readability: Can the result easily be read by humans?
- availability on different platforms: For which programming languages do libraries or bindings exist?
- security: Is it possible to decode bytes to arbitrary objects?

Which codec is the best very much depends on your specific requirements. An evaluation of different codecs and serialization formats is beyond the scope of this document, though.

1.5.1 Which codecs does aiomas support?

Aiomas implements the following codecs:

- *aiomas.codecs.JSON*
- *aiomas.codecs.MsgPack*
- *aiomas.codecs.MsgPackBlosc*

JSON

We chose JSON as default, because it is available through the standard library (no additional dependencies) and because it is relatively efficient (both, in terms of performance and serialization results). It is also widely used and supported as well as human readable.

MsgPack

The MsgPack codec can be more efficient but requires you to compile a C extension. For this reason, it is not enabled by default but available as an extra feature. To install it run:

```
$ pip install -U aiomas[mp]  # Install aiomas with MsgPack
$ # or
$ pip install -U aiomas msgpack-python
```

MsgPackBlosc

If you want to send long messages, e.g., containing large NumPy arrays, further compressing the results of MsgPack with Blosc can give you additional performance. To enable it, install:

```
$ pip install -U aiomas[mpb]  # Install aiomas with MsgPack-Blosc
$ # or
$ pip install -U aiomas msgpack-python blosc
```

Which codec should I use?

You should always start with the default JSON codec. It should usually be "good enough".

If your messages contain large chunks of binary data (e.g., serialized NumPy arrays), you should evaluate MsgPack, because it natively serializes objects to bytes.

MsgPackBlosc may yield better performance then MsgPack if your messages become very large and/or you really send *a lot* of messages. The codec can decrease the memory consumption of your program and reduce the time it takes to send a message.

Note: All codecs live in the `aiomas.codecs` package but, for your convenience, you can also import them directly from `aiomas`.

1.5.2 How do I use codecs?

As a normal user, you don't have to interact with codecs directly. You only need to pass the class object of the desired codec as a parameter to some functions and classes if you don't want to use the default.

1.5.3 Which object types can be (de)serialized?

All codecs bundled with aiomas support serializing the following types out of the box:

- `NoneType`
- `bool`
- `int`
- `float`
- `str`
- `list` / `tuple`
- `dict`

MsgPack and MsgPackBlosc also support `bytes`.

Note: JSON deserializes both, lists *and* tuples, to lists. MsgPack on the other hand deserializes them

to tuples.

RPC connections support serializing arbitrary objects with RPC routers which get deserialized to Proxies for the corresponding remote object. See rpc_router_serialization for details.

In addition, connections made by a `Container` support Arrow date objects.

1.5.4 How do I add serializers for additional object types?

All functions and classes that accept a *codec* parameter also accept an optional list of *extra_serializers*. The list must contain callables with the following signature: `callable() -> tuple(type, serialization_func, deserialisation_func)`.

The *type* is a class object. The serializer will be applied to all *direct* instances of that class but *not* to subclasses. This may change in the future, however. The only exception is a serializer for `object` which, if specified, serves as a fall-back for objects that couln't be serialized other ways (this is used by RPC connections to serialize objects with an RPC router).

The *serializer_func* is a callable with one argument – the object to be serialized – and needs to return an object that is serializable by the base codec (e.g., a *str*, *bytes* or *dict*).

The *deserializer_func* has the same signature, but the argument is the serialized object and the return value a deserialized equivalent of the original object. Usually, "equivalent" means "an object of the same type as the original", but objects with an RPC router, for example, get deserialized to proxies for the original objects in order to allow remote procedure calls on them.

Here is an example that shows how a serializer for NumPy arrays might look like. It will only work for the *MsgPack* and *MsgPackBlosc* codecs, because the dict returned by *_serialize_ndarray()* contains byte strings which JSON cannot handle:

```
import aiomas
import numpy as np

def get_np_serializer():
    """Return a tuple *(type, serialize(), deserialize())* for NumPy arrays
    for usage with an :class:`aiomas.codecs.MsgPack` codec.

    """
    return np.ndarray, _serialize_ndarray, _deserialize_ndarray


def _serialize_ndarray(obj):
    return {
        'type': obj.dtype.str,
        'shape': obj.shape,
        'data': obj.tostring(),
    }


def _deserialize_ndarray(obj):
    array = np.fromstring(obj['data'], dtype=np.dtype(obj['type']))
    return array.reshape(obj['shape'])


# Usage:
```

```
c = aiomas.Container(('localhost', 5555), codec=aiomas.MsgPack,
                     extra_serializers=[get_np_serializer])
```

1.5.5 How to create custom codecs

The base class for all codecs is *aiomas.codecs.Codec*.

Subclasses must at least implement the *encode()* and *decode()* methods.

You can use the existing codecs (e.g., *JSON* or *MsgPack*) as examples.

1.6 Container clocks

- Why clocks?
- What are clocks used for?
- Which clocks exists?
- What's their API?
- How to write custom codecs?

1.7 Testing and debugging

[Status: draft]

- asyncio's debug mode is honored. If it is activate, aiomas also falls into debug mode and gives you better / more detailed exceptions in some cases. This impacts performance, so it isn't activated always.

1.7.1 Testing coroutines with pytest

A naïve approach would be:

```
# tests/test_coros.py
import asyncio

def test_coro():
    loop = asyncio.get_event_loop()

    @asyncio.coroutine
    def do_test():
        yield from asyncio.sleep(0.1)
        assert 0  # onoes!

    loop.run_until_complete(do_test())
```

Creating and closing a loop should better be a fixture:

```
# tests/conftest.py
import asyncio


@pytest.yield_fixture
def loop():
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    yield loop
    loop.close()


# tests/test_coros.py
def test_coro(loop):
    @asyncio.coroutine
    def do_test():
        yield from asyncio.sleep(0.1)
        assert 0  # onoes!

    loop.run_until_complete(do_test())
```

Wouldn't it be cool if tests actually looked like this:

```
# tests/test_coros.py
@asyncio.coroutine
def test_coro(loop):
    yield from asyncio.sleep(0.1)
    assert 0
```

It's possible. You just have to create a small pytest plug-in:

```
# tests/conftest.py
import asyncio


@pytest.yield_fixture
def loop():
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    yield loop
    loop.close()


def pytest_pycollect_makeitem(collector, name, obj):
    """Collect asyncio coroutines as normal functions, not as generators."""
    if collector.funcnamefilter(name) and asyncio.iscoroutinefunction(obj):
        return list(collector._genfunctions(name, obj))


def pytest_pyfunc_call(pyfuncitem):
    """If ``pyfuncitem.obj`` is an asyncio coroutinefunction, execute it via
    the event loop instead of calling it directly."""
    testfunction = pyfuncitem.obj

    if not asyncio.iscoroutinefunction(testfunction):
        return

    # Copied from _pytest/python.py:pytest_pyfunc_call()
```

```
    funcargs = pyfuncitem.funcargs
    testargs = {}
    for arg in pyfuncitem._fixtureinfo.argnames:
        testargs[arg] = funcargs[arg]
    coro = testfunction(**testargs)  # Will not execute the test yet!

    # Run the coro in the event loop
    loop = testargs.get('loop', asyncio.get_event_loop())
    loop.run_until_complete(coro)

    return True
```

This is tested with pytest 2.6 and 2.7. Maybe newer releases of pytest will include something like this out-of-the-box.

1.8 API reference

The API reference provides detailed descriptions of aiomas' classes and functions.

1.8.1 aiomas

This module provides easier access to the most used components of *aiomas*. This purely for your convenience and you can, of cource, also import everything from its actual submodule.

Decorators

expose(func)	Decorator that enables RPC access to the decorated function.

Functions

async(coro_or_future[, ignore_cancel, loop])	Run asyncio.async() with *coro_or_future* and set a callback th

Exceptions

RemoteException(origin, remote_traceback)	Wraps a traceback of an exception on the other side of a channel.

Classes

Agent(container, addr)	Base class for all agents.
Container(addr[, clock, codec, ...])	Container for agents.
JSON()	A *Codec* that uses *JSON* to encode and decode messages.
MsgPack()	A *Codec* that uses *msgpack* to encode and decode messages.
MsgPackBlosc()	A *Codec* that uses *msgpack* to encode and decode messages and *blosc* to
AsyncioClock()	asyncio based real-time clock.
ExternalClock(utc_start[, init_time])	A clock that can be set by external process in order to synchronize it with

1.8.2 aiomas.agent

This module implements the base class for agents (*Agent*) and containers for agents (*Container*).

Every agent must live in a container. A container can contain one ore more agents. Containers are responsible for making connections to other containers and agents. They also provide a factory function for spawning new agent instances and registering them with the container.

Thus, the *Agent* base class is very light-weight. It only has a name, a reference to its container and an RPC router (see *aiomas.rpc*).

class aiomas.agent.**Container**(*addr*, *clock=None*, *codec=None*, *extra_serializers=()*)

Container for agents.

It can instantiate new agents via *spawn()* and can create connections to other agents (via *connect()*).

In order to destroy a container and close all of its sockets, call *shutdown()*.

When a container is created, it also creates a server socket and binds it to *addr* which is a `('host', port)` tuple (see the *host* and *port* parameters of `asyncio.BaseEventLoop.create_server()` for details).

You can optionally also pass a *codec* class. Note that containers can only "talk" to containers using the same codec.

You can also pass a list of *extra_serializers* for the codec. The list entires need to be callables that return a tuple with the arguments for *add_serializer()*.

To decouple a multi-agent system from the system clock, you can pass an optional *clock* object which should be an instance of *BaseClock*. This makes it easier to integrate your system with other simulators that may provide a clock for you or to let your MAS run as fast as possible. By default, *AsyncioClock* will be used.

codec

The codec used by this container. Instance of *aiomas.codecs.Codec*.

clock

The clock of the container. Instance of *aiomas.clocks.BaseClock*.

spawn(*agent_type*, **args*, ***kwargs*)

Create an instance of *agent_type*, passing a reference to this container, a name and the provided *args* and **kwargs** to it.

This is equivalent to `agent = agent_type(container, name, *args, **kwargs)`, but also registers the agent with the container.

connect(*url*)

Connect to the argent available at *url* and return a proxy to it.

url is a string `<protocol>://<addr>//<agent-id>` (e.g., `'tcp://localhost:5555/0'`).

shutdown(*async=False*)

Close the container's server socket and the RPC services for all outgoing TCP connections.

If *async* is left to `False`, this method calls `asyncio.BaseEventLoop.run_until_complete()` in order to wait until all sockets are closed.

If the event loop is already running when you call this method, set *async* to `True`. The return value then is a coroutine that you need to `yield from` in order to actually shut the container down:

```
yield from container.shutdown(async=True)
```

validate_aid(*aid*)
Return the class name for the agent represented by *aid* if it exists or `None`.

class aiomas.agent.**Agent**(*container*, *addr*)
Base class for all agents.

router
Descriptor that creates an RPC *Router* for every agent instance.

You can override this in a sub-class if you need to. (Usually, you don't.)

container
The *Container* that the agent lives in.

addr
The agent's address.

1.8.3 aiomas.channel

This module implements and asyncio `asyncio.Protocol` protocol for a request-reply *Channel*.

aiomas.channel.**open_connection**(*addr*, *, *loop=None*, *codec=None*, *extra_serializers=()*, ***kwds*)
Return a *Channel* connected to *addr*.

This is a convenience wrapper for `asyncio.BaseEventLoop.create_connection()` and `asyncio.BaseEventLoop.create_unix_connection()`.

If *addr* is a tuple `(host, port)`, a TCP connection will be created. If *addr* is a string, it should be a path name pointing to the unix domain socket to connect to.

You can optionally provide the event *loop* to use.

By default, the *JSON codec* is used. You can override this by passing any subclass of *aiomas.codecs.Codec* as *codec*.

You can also pass a list of *extra_serializers* for the codec. The list entires need to be callables that return a tuple with the arguments for *add_serializer()*.

The remaining keyword argumens *kwds* are forwarded to `asyncio.BaseEventLoop.create_connection()` and `asyncio.BaseEventLoop.create_unix_connection()` respectively.

aiomas.channel.**start_server**(*addr*, *client_connected_cb*, *, *loop=None*, *codec=None*, *extra_serializers=()*, ***kwds*)
Start a server listening on *addr* and call *client_connected_cb* for every client connecting to it.

This function is a convenience wrapper for `asyncio.BaseEventLoop.create_server()` and `asyncio.BaseEventLoop.create_unix_server()`.

If *addr* is a tuple `(host, port)`, a TCP socket will be created. If *addr* is a string, a unix domain socket at this path will be created.

The single argument of the callable *client_connected_cb* is a new instance of *Channel*.

You can optionally provide the event *loop* to use.

By default, the *JSON codec* is used. You can override this by passing any subclass of *aiomas.codecs.Codec* as *codec*.

You can also pass a list of *extra_serializers* for the codec. The list entires need to be callables that return a tuple with the arguments for *add_serializer()*.

The remaining keyword argumens *kwds* are forwarded to asyncio.BaseEventLoop.create_server() and asyncio.BaseEventLoop.create_unix_se respectively.

class aiomas.channel.**ChannelProtocol**(*codec*, *client_connected_cb=None*, ***, *loop*)
Asyncio asyncio.Protocol which connects the low level transport with the high level *Channel* API.

The *codec* is used to (de)serialize messages. It should be a sub-class of *aiomas.codecs.Codec*.

Optionally you can also pass a function/coroutine *client_connected_cb* that will be executed when a new connection is made (see *start_server()*).

connection_made(*transport*)
Create a new *Channel* instance for a new connection.

Also call the *client_connected_cb* if one was passed to this class.

connection_lost(*exc*)
Set a ConnectionError to the *Channel* to indicate that the connection is closed.

data_received(*data*)
Buffer incomming data until we have a complete message and then pass it to *Channel*.

Messages are fixed length. The first four bytes (in network byte order) encode the length of the following payload. The payload is a triple `(msg_type, msg_id, content)` encoded with the specified *codec*.

eof_received()
Set a ConnectionResetError to the *Channel*.

write(*content*)
Serialize *content* and write the result to the transport.

This method is a coroutine.

pause_writing()
Set the *paused* flag to `True`.

Can only be called if we are not already paused.

resume_writing()
Set the *paused* flat to `False` and trigger the waiter future.

Can only be called if we are paused.

class aiomas.channel.**Request**(*content*, *message_id*, *protocol*)
Represents a request returned by *Channel.recv()*. You shoudn't instantiate it yourself.

content contains the incoming message.

msg_id is the ID for that message. It is unique within a channel.

protocol is the channel's *ChannelProtocol* instance that is used for writing back the reply.

To reply to that request you can `yield from` *Request.reply()* or *Request.fail()*.

content
The content of the incoming message.

reply(*result*)
Reply to the request with the provided result.

fail(*exception*)
Indicate a failure described by the *exception* instance.

This will raise a *RemoteException* on the other side of the channel.

class aiomas.channel.**Channel**(*protocol*, *codec*, *transport*, *loop*)
A Channel represents a request-reply channel between two endpoints. An instance of it is returned by *open_connection()* or is passed to the callback of *start_server()*.

protocol is an instance of *ChannelProtocol*.

transport is an `asyncio.BaseTransport`.

loop is an instance of an `asyncio.BaseEventLoop`.

codec
The codec used to de-/encode messages send via the channel.

transport
The transport of this channel (see the Python documentation for details).

send(*content*)
Send a request *content* to the other end and return a future which is triggered when a reply arrives.

One of the following exceptions may be raised:

- *RemoteException*: The remote site raised an exception during the computation of the result.
- `ConnectionError` (or its subclass `ConnectionResetError`): The connection was closed during the request.
- `RuntimeError`:
  - If an invalid message type was received.
  - If the future returned by this method was already triggered or canceled by a third party when an answer to the request arrives (e.g., if a task containing the future is cancelled). You get more detailed exception messages if you enable asyncio's debug mode

```
try:
    result = yield from channel.request('ohai')
except RemoteException as exc:
    print(exc)
```

recv()
Wait for an incoming *Request* and return it.

May raise one of the following exceptions:

- `ConnectionError` (or its subclass `ConnectionResetError`): The connection was closed during the request.
- `RuntimeError`: If two processes try to read from the same channel or if an invalid message type was received.

close()
Close the channel's transport.

get_extra_info(*name*, *default=None*)
Wrapper for `asyncio.BaseTransport.get_extra_info()`.

1.8.4 aiomas.clocks

Clocks to be used with *aiomas.agent.Container*.

All clocks should subclass *BaseClock*. Currently available clock types are:

- *AsyncioClock*: a real-time clock synchronized with the `asyncio` event loop.
- *ExternalClock*: a clock that can be set by external tasks / processes in order to synchronize it with external systems or simulators.

class aiomas.clocks.**BaseClock**
Interface for clocks.

Clocks must at least implement *time()* and *utcnow()*.

time()
Return the value (in seconds) of a monotonic clock.

The return value of consecutive calls is guaranteed to be greater or equal then the results of previous calls.

The initial value may not be defined. Don't depend on it.

utcnow()
Return an `arrow.arrow.Arrow` date with the current time in UTC.

sleep(*dt*, *result=None*)
Sleep for a period *dt* in seconds. Return an `asyncio.Future`.

If *result* is provided, it will be passed back to the caller when the coroutine has finished.

sleep_until(*t*, *result=None*)
Sleep until the time *t*. Return an `asyncio.Future`.

t may either be a number in seconds or an `arrow.arrow.Arrow` date.

If *result* is provided, it will be passed back to the caller when the coroutine has finished.

call_in(*dt*, *func*, **args*)
Schedule the execution of `func(*args)` in *dt* seconds and return immediately.

Return an opaque handle which lets you cancel the scheduled call via its `cancel()` method.

call_at(*t*, *func*, **args*)
Schedule the execution of `func(*args)` at *t* and return immediately.

t may either be a number in seconds or an `arrow.arrow.Arrow` date.

Return an opaque handle which lets you cancel the scheduled call via its `cancel()` method.

class aiomas.clocks.**AsyncioClock**
asyncio based real-time clock.

class aiomas.clocks.**ExternalClock**(*utc_start*, *init_time=0*)
A clock that can be set by external process in order to synchronize it with other systems.

The initial UTC date *utc_start* may either be an arrow.arrow.Arrow instance or something that arrow.factory.ArrowFactory.get() can parse.

1.8.5 aiomas.codecs

This package imports the codecs that can be used for de- and encoding incoming and outgoing messages:

- *JSON* uses JSON
- *MsgPack* uses msgpack
- *MsgPackBlosc* uses msgpack and Blosc

All codecs should implement the base class *Codec*.

class aiomas.codecs.**Codec**
Base class for all Codecs.

Subclasses must implement *encode()* and *decode()*.

encode(*data*)
Encode the given *data* and return a bytes object.

decode(*data*)
Decode *data* from bytes to the original data structure.

add_serializer(*type*, *serialize*, *deserialize*)
Add methods to *serialize* and *deserialize* objects typed *type*.

This can be used to de-/encode objects that the codec otherwise couldn't encode.

serialize will receive the unencoded object and needs to return an encodable serialization of it.

deserialize will receive an objects representation and should return an instance of the original object.

serialize_obj(*obj*)
Serialize *obj* to something that the codec can encode.

deserialize_obj(*obj_repr*)
Deserialize the original object from *obj_repr*.

class aiomas.codecs.**JSON**
A *Codec* that uses *JSON* to encode and decode messages.

class aiomas.codecs.**MsgPack**
A *Codec* that uses *msgpack* to encode and decode messages.

class aiomas.codecs.**MsgPackBlosc**
A *Codec* that uses *msgpack* to encode and decode messages and *blosc* to compress them.

aiomas.codecs.**serializable**(*repr=True*)
Class decorator that makes the decorated class serializable by *aiomas.codecs*.

The decorator tries to extract all arguments to the class' `__init__()`. That means, the arguments must be available as attributes with the same name.

The decorator adds the following methods to the decorated class:

- `__asdict__()`: Returns a dict with all __init__ parameters
- `__fromdict__(dict)`: Creates a new class instance from *dict*
- `__serializer__()`: Returns a tuple with args for *Codec.add_serializer()*
- `__repr__()`: Returns a generic instance representation. Adding this method can be deactivated by passing `repr=False` to the decorator.

Example:

```
>>> import aiomas.codecs
>>>
>>> @aiomas.codecs.serializable
... class A:
...     def __init__(self, x, y):
...         self.x = x
...         self._y = y
...
...     @property
...     def y(self):
...         return self._y
>>>
>>> codec = aiomas.codecs.JSON()
>>> codec.add_serializer(*A.__serializer__())
>>> a = codec.decode(codec.encode(A(1, 2)))
>>> a
A(x=1, y=2)
```

1.8.6 aiomas.exceptions

Exception types used by *aiomas*.

exception `aiomas.exceptions.`**`RemoteException`**(*origin*, *remote_traceback*)

Wraps a traceback of an exception on the other side of a channel.

origin is the remote peername.

remote_traceback is the remote exception's traceback.

1.8.7 aiomas.rpc

This module implements remote procedure calls (RPC) on top of request-reply channels (see *aiomas.channel*).

RPC connections are represented by instances of *RpcClient* (one for each side of a *aiomas.channel.Channel*). They provide access to the functions served by the remote side of the channel via *Proxy* instances. Optionally, they can provide their own RPC service (via *rpc_service()*) so that the remote side can make calls as well.

An RPC service is defined by a *Router*. A router resolves paths requested by the remote side. It can also handle sub-routers (which allows you to build hierarchies for nested calls) and is able to perform a reverse-lookup of a router (mapping a fuction to its path).

Routers provide the functions and methods of dictionaries or class instances. Dict routers can be created by passing a dictionary to `Router`. For classes, you create a `Service` instance as *router* class attribute. This creates a *Descriptor* which then creates a new router instance for each class instance.

Functions that should be callable from the remote side must be decorated with `expose()`; `Router.expose()` and `Service.expose()` are aliases for it.

aiomas.rpc.**open_connection**(*addr*, *, *router=None*, ***kwds*)
Return an `RpcClient` connected to *addr*.

This is a convenience wrapper for `aiomas.channel.open_connection()`. All keyword arguments *(kwds)* are forwared to it.

You can optionally pass a *router* to allow the remote site to call back to us.

aiomas.rpc.**start_server**(*addr*, *router*, *client_connected_cb=None*, ***kwds*)
Start a server socket on *host:port* and create an RPC service with the provided *handler* for each new client.

This is a convenience wrapper for `aiomas.channel.start_server()`. All keyword arguments *(kwds)* are forwared to it.

router must be a `Router` instance for the `rpc_service()` that is started for each new connection.

client_connected_cb is an optional callback that will be called with with the `RpcClient` instance for each new connection.

Raise a `ValueError` if *handler* is not decorated properly.

aiomas.rpc.**rpc_service**(*router*, *channel*)
Serve the functions provided by the `Router` *router* via the `Channel` *channel*.

Forward errors raised by the handler to the caller.

Stop running when the connection closes.

aiomas.rpc.**expose**(*func*)
Decorator that enables RPC access to the decorated function.

func will not be wrapped but only gain an `__rpc__` attribute.

class aiomas.rpc.**RoutingDict**(*dict=None*)
Wrapper for dicts so that they can be used as RPC routers.

dict = None
The wrapped dict.

router = None
The dict's router instance.

class aiomas.rpc.**Router**(*obj*)
The Router resolves paths to functions provided by their object *obj* (or its children). It can also perform a reverse lookup to get the path of the router (and the router's *obj*).

The *obj* can be a class, an instance or a dict.

obj = None
The object to which this router belongs to.

name = None
The name of the router (empty for root routers).

parent = None
The parent router or `None` for root routers.

path
The path to this router (without trailing slash).

resolve (*path*)
Resolve *path* and return the corresponding function.

path is a string with path components separated by / (without trailing slash).

Raise a `LookupError` if no handler function can be found for *path* or if the function is not exposed (see *expose()*).

static expose (*func*)
Alias for *expose()*.

add (*name*)
Add the sub-router *name* (stored at `self.obj.<name>`) to this router.

Convenience wrapper for *set_sub_router()*.

set_sub_router (*router*, *name*)
Set *self* as parent for the *router* named *name*.

class aiomas.rpc.**Service** (*sub_routers=()*)
A Data Descriptor that creates a new *Router* instance for each class instance to which it is set.

The attribute name for the Service should always be *router*:

```
class Spam:
    router = aiomas.rpc.Service()
```

You can optionally pass a list with the attribute names of classes with sub-routers. This required to build hierarchies of routers, e.g.:

```
class Eggs:
    router = aiomas.rpc.Service()


class Spam:
    router = aiomas.rpc.Service(['eggs'])

    def __init__(self):
        self.eggs = Eggs()  # Instance with a sub-router
```

static expose (*func*)
Alias for *expose()*.

class aiomas.rpc.**RpcClient** (*channel*, *router=None*)
The RpcClient provides proxy objects for remote calls via its *remote* attribute.

channel is a *Channel* instance for communicating with the remote side.

If *router* is not `None`, it will also start its own RPC service so the other side can make calls to us as well.

channel
The communication *Channel* of this instance.

service
The RPC service process for this connection.

remote
A *Proxy* for remote methods.

close()
Close the connection.

class aiomas.rpc.**Proxy**(*channel*, *path*)
Proxy object for remote objects and functions.

__weakref__
list of weak references to the object (if defined)

__getattr__(*name*)
Return a new proxy for *name*.

__call__(**args*, ***kwargs*)
Call the remote method represented by this proxy and return its result.

The result is a future, so you need to `yield from` it in order to get the actual return value (or exception).

1.8.8 aiomas.util

This module contains some utility functions.

aiomas.util.**async**(*coro_or_future*, *ignore_cancel=True*, *loop=None*)
Run `asyncio.async()` with *coro_or_future* and set a callback that instantly raises all exceptions.

If *ignore_cancel* is left `True`, no exception is raised if the task was canceled. If you also want to raise the `CancelledError`, set the flag to `False`..

Return an `asyncio.Task` object.

The difference between this function and `asyncio.async()` subtle, but important if an error is raised by the task:

`asyncio.async()` returns a future (`asyncio.Task` is a subclass of `asyncio.Future`) for the task that you created. By the time that future goes out of scope, asyncio checks if someone was interested in its result or not. If the result was never retrieved, the exception is printed to *stderr*.

If you call it like `asyncio.async(mytask())` (note that we don't keep a reference to the future here), an exception in *mytask* will pre printed immediately when the task is done. If, however, we store a reference to the future (`fut = asyncio.async(mytask())`), the exception only gets printed when `fut` goes out of scope. That means if, for example, an *Agent* creates a task and stores it as an instance attribute, our system may keep running for a long time after the exception has occured (or even block forever) and we won't see any stacktrace. This is because the reference to the task is still there and we could, in theory, still retrieve the exception from there.

Since this can make debugging very hard, this method simply registers a callback to the future. The callback will try to get the result from the future when it is done and will thus print any exceptions immediately.

aiomas.util.**run**(*until=None*)

Run the event loop forever or until the task/future *until* is finished.

This is an alias to asyncio's `run_forever()` if *until* is `None` and to `run_until_complete()` if not.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

a

Symbols

A

B

C

D

E

F

G

J

M

N

O

P

R

S

T

U

V

W