aiomas Documentation

Release 0.6.1

Stefan Scherfke

November 21, 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 aiomas
>>>
>>> class TestAgent(aiomas.Agent):
...     def __init__(self, container):
...         super().__init__(container)
...         print('Ohai, I am %s' % self)
...
...     async def run(self, addr):
...         remote_agent = await self.container.connect(addr)
...         ret = await 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.create(('localhost', 5555))
>>> agents = [TestAgent(c) 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.

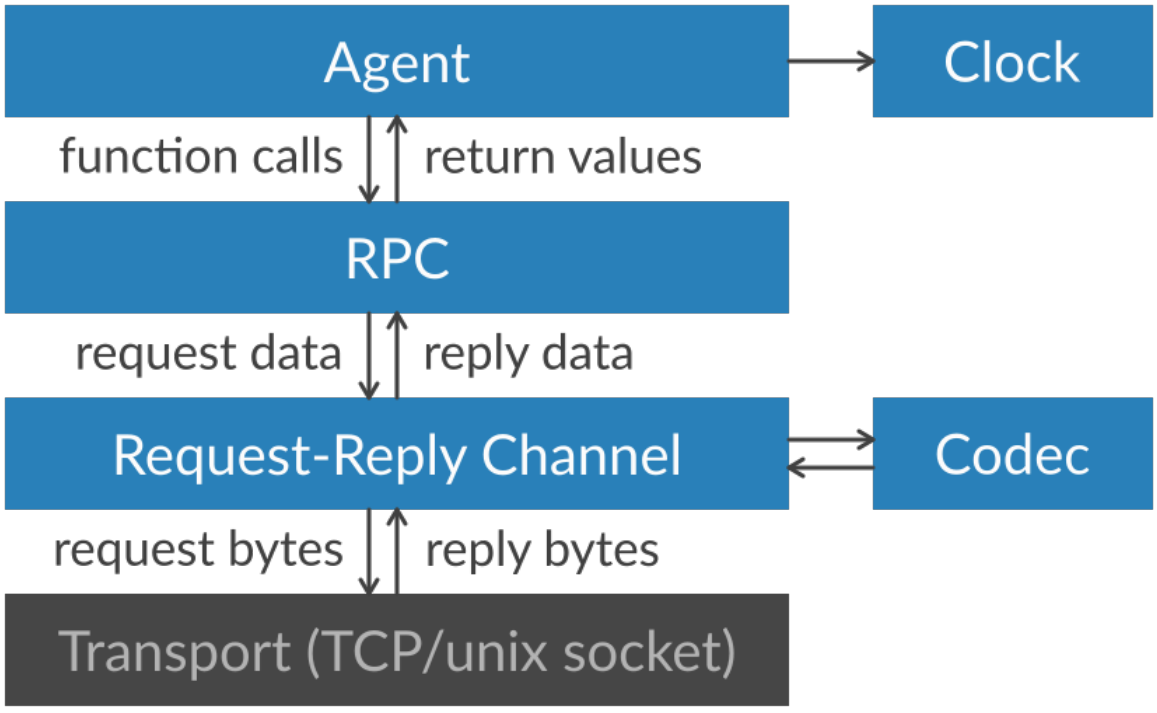
CHAPTER 1

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 = await 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 = await 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).

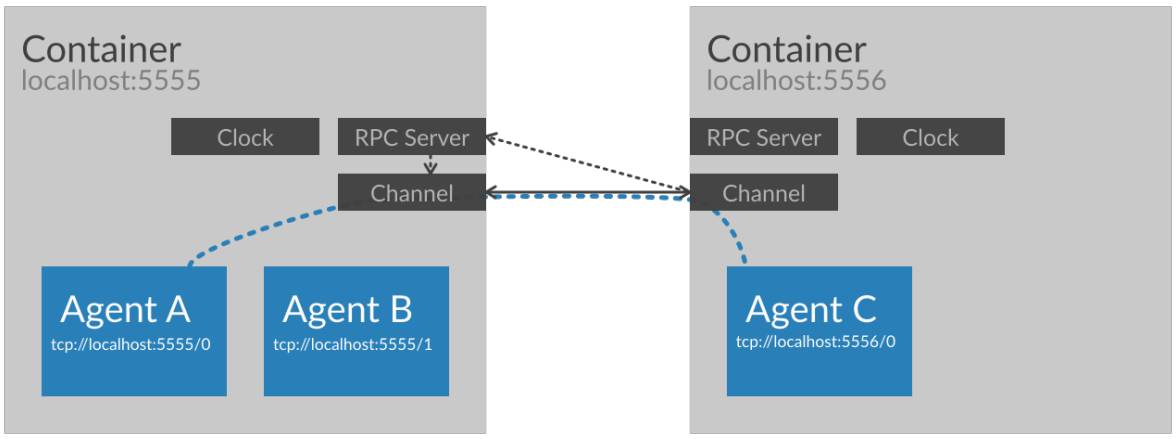
You can also imagine them as being like normal objects that call other object's methods. But instead of calling these methods directly, they do remote procedure calls (RPC) via a network connection.

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!":

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

Agents should be a subclass of *Agent*. They always need a reference to the container they live in, so that `Agent.__init__()` can register the agent with that container. If you overide `__init__()`, always make sure to call `super().__init__(container)` from your own implementation.

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.create(('localhost', 5555))
>>> agent = HelloWorld(container, '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.

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):
...
...     async def run(self, callee_addr):
...         print(self, 'connecting to', callee_addr)
...         callee = await self.container.connect(callee_addr)
...         print(self, 'connected to', callee)
...         result = await callee.spam(3)
...         print(self, 'got', result)
>>>
>>>
>>> container = aiomas.Container.create(('localhost', 5555))
>>> callee = Callee(container)
>>> caller = Caller(container)
>>> 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 `await` 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 `await` 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

Here is a minimal example that shows how the *`Channel`* can be used:

```
>>> import aiomas
>>>
>>>
>>> ADDR = ('localhost', 5555)
>>>
>>>
>>> async def handle_client(channel):
```

```
...     req = await channel.recv()
...     print(req.content)
...     await req.reply('cya')
...     channel.close()
>>>
>>>
>>> async def client():
...     channel = await aiomas.channel.open_connection(ADDR)
...     rep = await channel.send('ohai')
...     print(rep)
...     channel.close()
>>>
>>>
>>> server = aiomas.run(aiomas.channel.start_server(ADDR, handle_client))
>>> aiomas.run(client())
ohai
cya
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

1.4.1 How can I bind a server socket to a random port?

You cannot ask your OS for an available port but have to try a randomly chosen port until you succeed:

```
>>> import random
>>>
>>> max_tries = 100
>>> port_range = (49152, 65536)
>>>
>>> async def random_server(host, port_range, max_tries):
...     for i in range(max_tries):
...         try:
...             port = random.randrange(*port_range)
...             server = await aiomas.channel.start_server(
...                 (host, port), handle_client)
...         except OSError as oe:
...             if oe.errno != 48:
...                 # Re-raise if not errno 48 ("address already in use")
...                 raise
...         else:
...             return server, port
...     raise RuntimeError('Could not bind server to a random port.')
>>>
>>> server, port = aiomas.run(random_server('localhost', port_range, max_tries))
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

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()

    async def do_test():
        await 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):
    async def do_test():
        await 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
async def test_coro(loop):
    await 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 Enabling transport security (TLS)

This guide explains how you can encrypt all messages sent with aiomas. Transport layer security (TLS, formerly known as SSL) can be applied in a similar fashion to all three layers (channel, RPC, agent) of aiomas and the following sections will show you how.

Note: Even if you don't have much experience with cryptography, you should be able to follow this guide and use TLS encryption for your program.

Nonetheless, I strongly recommend you to learn the basics of it. A good read is Crypto 101, by Laurens Van Houtven. Sean Cassidy also provides a nice overview about starting with crypto. There are also various tutorials for setting up your own PKI (1, 2, 3, 4).

1.8.1 Security architecture

This guide assumes that your system is self-contained and you control all parts of it. This allows you to use TLS 1.2 with a modern cipher and to setup a public key infrastructure (PKI) with a self-signed root CA. All machines that you deploy your system on only thrust that CA (and ignore the CAs bundled with your OS or web browser).

Ideally, the root CA should be created on separate, non-production machine. Depending on your security requirements, that machine should not even be connected to the network.

You create a certificate signing request (CSR) on each production machine. You copy the CSR to your root CA which signs it. You then copy the signed certificate back to the production machine. Ideally, you should use an SD card for this (they are more secure than USB flash drives), but again, this depends on your security requirements and using SSH might also work for you.

1.8.2 The root CA

First, you create the root CA's private key. It should at least be 2048, or better, 4096 bits long. It should also be encrypted with a strong passphrase:

```
$ openssl genrsa -aes256 -out ca.key 4096
```

The key should never leave the machine, except if you store it somewhere save (e.g., on an SD card).

Now you sign the key and create the root certificate. You use it together with the private key for signing CSRs for other machines:

```
$ openssl req -new -x509 -nodes -key ca.key -out ca.pem -days 1000
```

The command above requires some input from you. The *Common Name* (e.g., the FQDN) that you associate with the certificate must be different from the ones that you use for your production machine's CSRs. The certificate should be valid for a longer period of time than the CSRs that it signs.

1.8.3 Certificates for production machines

You need to create one private key and CSR on each of your production machines:

```
$ openssl genrsa -out device.key 4096
$ openssl req -new -key device.key -out device.csr
```

This time, the private key is not encrypted. Otherwise, you'd have to hard-code the password into your source code (which would make the encryption futile) or enter it each time you start your program (which is unfeasible for a distributed multi-agent system). The private key should still not leave the machine; so don't even think of putting it into version control or reusing it on another machine.

The CSR creation requires similar input as the CA certificate that you created above. As *Common Name* or *FQDN* you should enter the address on which the machines server socket will be listening.

Copy `device.csr` to the root CA machine and sign it there:

```
$ openssl x509 -CA ca.pem -CAkey ca.key -CAcreateserial -req -in device.csr -out device.
```

The certificate will be valid for one year. You can change this if you want.

Transfer the certificate `device.pem` as well as copy of the CA certificate `ca.pem` back to the originating machine.

The `device.pem` will be used to authenticate that machine against other machines. `ca.pem` will be used to verify other machine's certificates when they try to authenticate themselves.

1.8.4 Enabling TLS for channels and RPC connections

In pure *asyncio* programs, you enable SSL/TLS by passing an `ssl.SSLContext` instance to `create_connection()` and `create_server()`.

`aiomas.channel.open_connection()` and *`aiomas.channel.start_server()`* (and similarly in the *`aiomas.rpc`* module) are just wrappers for the corresponding asyncio methods and will forward an `SSLContext` to them if one is provided.

Here is a minimal, commented example that demonstrate how to create proper SSL contexts:

```
>>> import asyncio
>>> import ssl
>>>
>>> import aiomas
>>>
>>>
>>> async def client(addr, ssl):
...     """Connect to *addr* and use the *ssl* context to enable TLS.
...     Send "ohai" to the server, print its reply and terminate."""
...     channel = await aiomas.channel.open_connection(addr, ssl=ssl)
...     reply = await channel.send('ohai')
...     print(reply)
...     channel.close()
>>>
>>>
>>> async def handle_client(channel):
...     """Handle client requests by printing them.  Send a reply and
...     terminate."""
...     request = await channel.recv()
...     print(request.content)
...     await request.reply('cya')
...     channel.close()
>>>
>>>
>>> addr = ('127.0.0.1', 5555)
>>>
>>> # Create an SSLContext for the server supporting (only) TLS 1.2 with
```

```
>>> # Eliptic Curve Diffie-Hellman and AES in Galois/Counter Mode
>>> server_ctx = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
>>> server_ctx.set_ciphers('ECDH+AESGCM')
>>> # Load the cert and key for authentication against clients
>>> server_ctx.load_cert_chain(certfile='device.pem', keyfile='device.key')
>>> # The client also needs to authenticate itself with a cert signed by ca.pem
>>> server_ctx.verify_mode = ssl.CERT_REQUIRED
>>> server_ctx.load_verify_locations(cafile='ca.pem')
>>> # Only use ECDH keys once per SSL session
>>> server_ctx.options |= ssl.OP_SINGLE_ECDH_USE
>>> # Disable TLS compression
>>> server_ctx.options |= ssl.OP_NO_COMPRESSION
>>>
>>> # Start the server.
>>> # It will use "server_ctx" to enable TLS for each connection.
>>> server = aiomas.run(aiomas.channel.start_server(addr, handle_client,
...                                                 ssl=server_ctx))
>>>
>>> # Create an SSLContext for the client supporting (only) TLS 1.2 with
>>> # Eliptic Curve Diffie-Hellman and AES in Galois/Counter Mode
>>> client_ctx = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
>>> client_ctx.set_ciphers('ECDH+AESGCM')
>>> # The server needs to authenticate itself with a cert signed by ca.pem.
>>> # And we also want ot verify its hostname.
>>> client_ctx.verify_mode = ssl.CERT_REQUIRED
>>> client_ctx.load_verify_locations(cafile='ca.pem')
>>> client_ctx.check_hostname = True
>>> # Load the cert and key for authentication against the server
>>> client_ctx.load_cert_chain(certfile='device.pem', keyfile='device.key')
>>>
>>> # Run the client.  It will use "client_ctx" to enable TLS.
>>> aiomas.run(client(addr, client_ctx))
ohai
cya
>>>
>>> # Shutdown the server
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

As you can see, the SSL contexts used by servers and clients are slightly different. Clients should verify that the hostname they connected to is the same as in the server's certificate. Servers on the other hand can set a few more options for a TLS connection.

aiomas offers two functions that create secure SSL contexts with the same settings as in the example above – *make_ssl_server_context()* and *make_ssl_client_context()*:

```
>>> server_ctx = aiomas.make_ssl_server_context('ca.pem', 'device.pem', 'device.key')
>>> server = aiomas.run(aiomas.channel.start_server(
...     addr, handle_client, ssl=server_ctx))
>>>
>>> client_ctx = aiomas.make_ssl_client_context('ca.pem', 'device.pem', 'device.key')
>>> aiomas.run(client(addr, client_ctx))
ohai
cya
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

1.8.5 TLS configuration for agent containers

An agent *Container* has its own server socket and creates a number of client sockets when it connects to other containers.

You can easily enable TLS for both socket types by passing an *SSLCerts* instance to the container. This is a named tuple with the filenames of the root CA certificate, the certificate for authenticating the container as well as the corresponding private key:

```
>>> import aiomas
>>>
>>> sslcerts = aiomas.SSLCerts('ca.pem', 'device.pem', 'device.key')
>>> c = aiomas.Container.create(('127.0.0.1', 5555), ssl=sslcerts)
>>>
>>> # Start agents and run your system
>>> # ...
>>>
>>> c.shutdown()
```

The container will use the *make_ssl_server_context()* and *make_ssl_client_context()* functions to create the necessary SSL contexts.

If you need more flexibility, you can alternatively pass a tuple with two SSL contexts (one for the server and one for client sockets) to the container:

```
>>> import aiomas
>>>
>>> server_ctx = aiomas.make_ssl_server_context('ca.pem', 'device.pem', 'device.key')
>>> client_ctx = aiomas.make_ssl_client_context('ca.pem', 'device.pem', 'device.key')
>>> c = aiomas.Container.create(('127.0.0.1', 5555), ssl=(server_ctx, client_ctx))
>>>
>>> # Start agents and run your system
>>> # ...
>>>
>>> c.shutdown()
```

1.9 API reference

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

1.9.1 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.
serializable([cls, repr])	Class decorator that makes the decorated class serializable by *aiomas.codecs*.

Functions

async(coro_or_future[, ignore_cancel, loop])	Run `asyncio.async()` with *coro_or_future* and set a callb
run([until])	Run the event loop forever or until the task/future *until* is finish
make_ssl_server_context(cafile, certfile, ...)	Return an `ssl.SSLContext` that can be used by a server soc
make_ssl_client_context(cafile, certfile, ...)	Return an `ssl.SSLContext` that can be used by a client soc

Exceptions

AiomasError	Base class for all exceptions defined by aiomas.
RemoteException(origin, remote_traceback)	Wraps a traceback of an exception on the other side of a channel.

Classes

Agent(container)	Base class for all agents.
Container(base_url, clock, connect_kwargs)	Container for agents.
SSLCerts(cafile, certfile, keyfile)	`namedtuple()` storing the names of a CA file, a
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 *bl*
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

1.9.2 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.**SSLCerts**(*cafile*, *certfile*, *keyfile*)

`namedtuple()` storing the names of a CA file, a certificate file and the associated private key file.

See also *aiomas.util.make_ssl_server_context()* and *aiomas.util.make_ssl_client_context()*.

cafile
Alias for field number 0

certfile
Alias for field number 1

keyfile
Alias for field number 2

class aiomas.agent.**Container**(*base_url*, *clock*, *connect_kwargs*)

Container for agents.

You should not instantiate containers directly but use the *create()* method/coroutine instead. This makes sure that the container's server socket is fully operational when it is created.

The container allows its agents to create connections to other agents (via *connect()*).

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

classmethod create(*addr*, ***, *clock=None*, *codec=None*, *extra_serializers=None*, *ssl=None*, *async=False*)

Instantiate a container and create a server socket for it.

This function is a classmethod and coroutine.

Parameters

- **addr** – is the address that the server socket is bound to. It may be a `(host, port)` tuple or a path for a Unix domain socket.

  If `host` is `'0.0.0.0'`/`'::'`, the server is bound to all available IPv4 or IPv6 interfaces respectively. If `host` is `None` or `''`, the server is bound to all available IPv4 and IPv6 interfaces. In these cases, the machine's FQDN (see `socket.getfqdn()`) should be resolvable and point to that machine as it will be used for the agent's addresses.

  If `host` is a simple (IPv4 or IPv6) IP address, it will be used for the agent's addresses as is.

- **clock** – can be an instance of *BaseClock*.

  It allows you to decouple the container's (and thus, its agent's) time from the system clock. 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, the real-time *AsyncioClock* will be used.

- **codec** – can be a *Codec* subclass (not an instance!). *JSON* is used by default.

- **extra_serializers** – is an optional list of extra serializers for the codec. The list entries need to be callables that return a tuple with the arguments for *add_serializer()*.

- **ssl** – allows you to enable TLS for all incoming and outgoing TCP connections. It may either be an *SSLCerts* instance or a tuple containing two `SSLContext` instances, where the first one will be used for the server socket, the second one for client sockets.

- **async** – must be set to `True` if the event loop is already running when you call this method. This function then returns a coroutine that you need to `yield from` in order to get the container. By default it will block until the server has been started and return the container.

Returns a fully initialized *Container* instance if *async* is `False` or else a coroutine returning the instance when it is done.

Invocation examples:

```
# Synchronous:
container = Container.create(...)
```

```
        # Asynchronous:
        container = yield from Container.create(..., async=True)
```

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

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*)
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.

1.9.3 aiomas.channel

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

aiomas.channel.**DEFAULT_CODEC**
Default codec: *JSON*

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()`.

1.9.4 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.

class aiomas.clocks.**TimerHandle**(*future*, *callback*)
This class lets you cancel calls scheduled by *ExternalClock*.

cancel()
Cancel the scheduled call represented by this handle.

1.9.5 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*.

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)
```

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.

1.9.6 aiomas.exceptions

Exception types used by *aiomas*.

exception aiomas.exceptions.**AiomasError**
Base class for all exceptions defined 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.

1.9.7 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).

1.9.8 aiomas.util

This module contains some utility functions.

aiomas.util.**arrow_serializer**()
Return a serializer for *arrow* dates.

The return value is an argument tuple for *aiomas.codecs.Codec.add_serializer()*.

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.

aiomas.util.**make_ssl_server_context**(*cafile*, *certfile*, *keyfile*)
Return an ssl.SSLContext that can be used by a server socket.

The server will use the certificate in *certfile* and private key in *keyfile* (both in PEM format) to authenticate itself.

It requires clients to also authenticate themselves. Their certificates will be validated with the root CA certificate in *cafile*.

It will use *TLS 1.2* with *ECDH+AESGCM* encryption. ECDH keys won't be reused in distinct SSL sessions. Compression is disabled.

aiomas.util.**make_ssl_client_context**(*cafile*, *certfile*, *keyfile*)
Return an ssl.SSLContext that can be used by a client socket.

It uses the root CA certificate in *cafile* to validate the server's certificate. It will also check the server's hostname.

The client will use the certificate in *certfile* and private key in *keyfile* (both in PEM format) to authenticate itself.

It will use *TLS 1.2* with *ECDH+AESGCM* encryption.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

a

Symbols

A

B

C

D

E

F

G

J

K

M

N

O

P

R

S

T

U

V

W