aiomas Documentation

Release 1.0.0

Stefan Scherfke

Apr 18, 2016

Contents

PyPI | Bitbucket | Mailing list | IRC: #asyncio

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

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

Below you'll find a list of features. You can also take a look at the *overview section* to learn what aiomas is and see some simple examples. If you like this package, go and *install* it!

CHAPTER 1

Features

- Three layers of abstraction around raw TCP / unix domain sockets:
    1. Request-reply channels
    2. Remore-procedure calls (RPC)
    3. Agents and containers
- TLS support for authorization and encrypted communication.
- Interchangeable and extensible codecs: JSON and MsgPack (the latter optionally compressed with Blosc) are built-in. You can add custom codecs or write (de)serializers for your own objects to extend a codec.
- Deterministic, emulated sockets: A *LocalQueue* transport lets you send and receive message in a deterministic and reproducible order within a single process. This helps testing and debugging distributed algorithms.

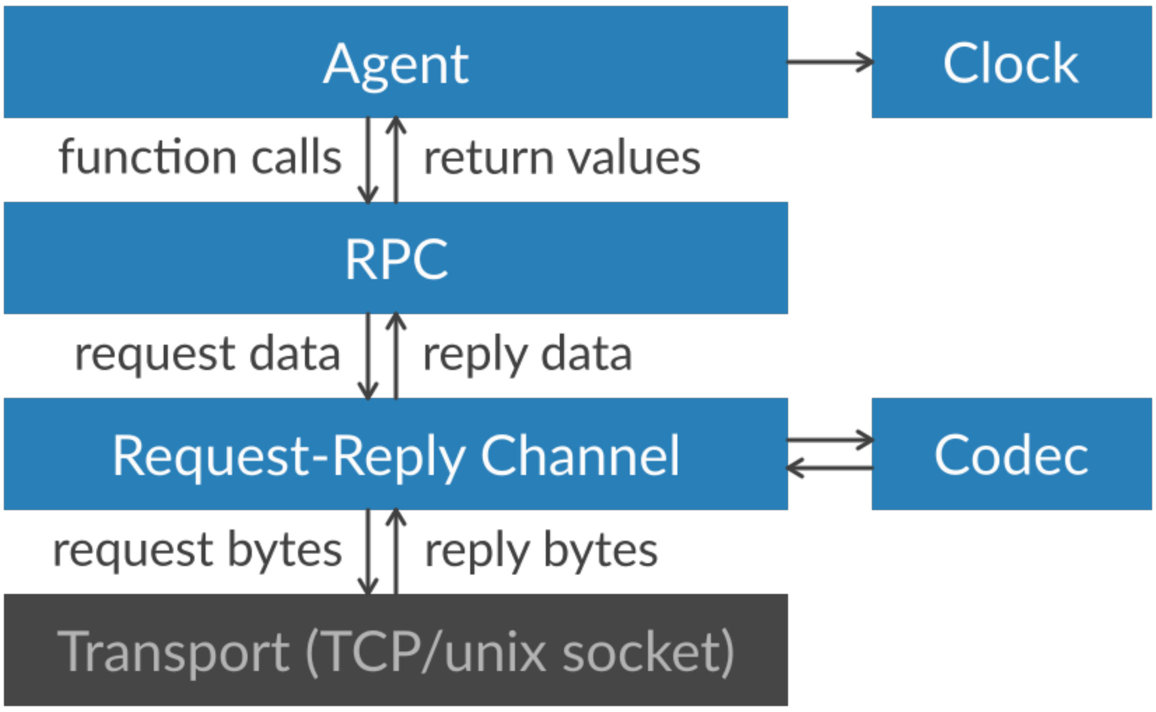
CHAPTER 2

Contents:

2.1 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 of numbers instead of single bytes.

   The `Channel` class lets you make *requests* and asynchronously wait for the corresponding *replies*.

   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.

```
>>> import aiomas
>>>
>>>
>>> async def handle_client(channel):
...     """Handle a client connection."""
...     req = await channel.recv()
...     print(req.content)
...     await req.reply('cya')
...     await channel.close()
>>>
>>>
>>> async def client():
...     """Client coroutine: Send a greeting to the server and wait for a
...     reply."""
...     channel = await aiomas.channel.open_connection(('localhost', 5555))
...     rep = await channel.send('ohai')
...     print(rep)
...     await channel.close()
>>>
>>>
>>> server = aiomas.run(aiomas.channel.start_server(('localhost', 5555), handle_cli
>>> aiomas.run(client())
ohai
cya
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

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 normal functions within a dict. On the peer side of the connection, proxy objects represent these exposed functions.

```
>>> import aiomas
>>>
>>>
>>> class MathServer:
...     router = aiomas.rpc.Service()
...
...     @router.expose
...     def add(self, a, b):
...         return a + b
...
>>>
>>> async def client():
...     """Client coroutine: Call the server's "add()" method."""
...     rpc_con = await aiomas.rpc.open_connection(('localhost', 5555))
...     rep = await rpc_con.remote.add(3, 4)
...     print('What's 3 + 4?', rep)
...     await rpc_con.close()
>>>
>>> server = aiomas.run(aiomas.rpc.start_server(('localhost', 5555), MathServer()))
>>> aiomas.run(client())
What's 3 + 4? 7
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

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 handling 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., external simulators).

```
>>> 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:555
>>> c.shutdown()
```

The following sections explain theses layers in more detail.

2.2 Installation

aiomas requires Python >= 3.4 and runs on Linux, OS X and Windows. The default installation uses the *JSON* codec and only has pure Python dependencies.

If you have an active virtualenv, you can just run pip to install it:

```
$ pip install aiomas
```

If you don't use a virtualenv (you should) and are not sure, which Python interpreter pip will use, you can manually select one:

```
$ python3.5 -m pip install aiomas
```

2.2.1 Updating aiomas

To upgrade your installation, use the `-U` flag for the `install` command:

```
$ pip install -U aiomas
```

2.2.2 Using MsgPack and Blosc

The MsgPack codec and its Blosc compressed version are optional features, that you need to explicitly install if you need them. Both packages require a C compiler for the installation:

```
$ pip install aiomas[mp]    # Enables the MsgPack codec
$ pip install aiomas[mpb]   # Enables the MsgPack and MsgPackBlosc codecs
```

Windows users can download pre-compiled binary packages from Christoph Gohlke's website (msgpack | blosc) and install them with *pip*:

```
C:\> pip install aiomas
C:\> pip install Downloads\msgpack_python-0.4.7-cp35-none-win_amd64.whl
C:\> pip install Downloads\blosc-1.2.8-cp35-none-win_amd64.whl
```

2.3 Topical Guides

2.3.1 The agent layer

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

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).

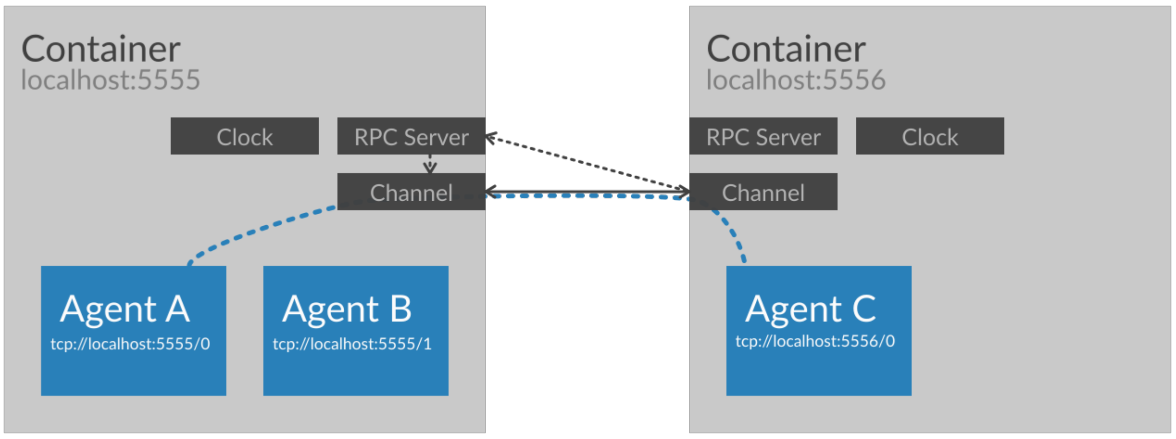
You can also imagine them as being like normal objects that call other object's methods. But instead of calling these methods directly, they do remote procedure calls (RPC) via a network connection.

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 provides 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:5555/42*.

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 for the moment. We'll come back to it later.

Components of a distributed system in aiomas

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* (that just means that your agent class exposes some methods and has one or more tasks running).

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.

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):
...         # We must pass a ref. to the container to "aiomas.Agent":
...         super().__init__(container)
...         self.name = name  # Our agent's name
...
...     async def run(self):
...         # This method defines the task that our agent will perform.
...         # It's usually called "run()" but you can name it as wou want.
...         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`. This base class needs a reference to the container the agents live in, so you must forward a *container* argument to it if you override `__init__()`.

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 task `run()` can either be started automatically in the agent's `__init__()` or manually after the agent has been instantiated. In our example, we will do the latter.

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).

Now lets see how we can instantiate and run our agent:

```
>>> # Containers need to be started via a factory function:
>>> container = aiomas.Container.create(('localhost', 5555))
>>>
>>> # Now we can instantiate an agent an start its task:
>>> agent = HelloWorld(container, 'Monty')
>>> aiomas.run(until=agent.run())
Monty says:
Hello, World!
Hello, World!
Hello, World!
>>> container.shutdown()  # Close all connections and shutdown the server
```

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 just a wrapper for `loop = asyncio.get_event_loop(); loop.run_until_complete(task)`.

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

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 clearness, 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):
...     # This agent does not need to override "__init__()".
...
...     # "expose"d methods can be called by other agents:
...     @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)
...         # Ask the container to make a connection to the other agent:
...         callee = await self.container.connect(callee_addr)
...         print(self, 'connected to', callee)
...         # "callee" is a proxy to the other agent.  It allows us to call
...         # the exposed methods:
...         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*. Exposed methods can be normal functions or 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` its result. 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.

Distributing agents over multiple containers

One container can house a (theoretically) unlimited number of agents. As long as your agents spent most of the time waiting for network IO, there's no need to use more than one container.

If you notice, that the Python process with your program fully utilizes its CPU core (remember, pure Python only uses one core), its time to spawn sub-processes with its own container to actually parallelize your application. The `aiomas.subproc` module provides some helpers for this use case.

Running multiple agent containers in a single process might only be helpful for demonstration or debugging purposes. In the latter case, you should also take a look at the `aiomas.local_queue` transport. You can replace normal TCP sockets with it and gain a deterministic order of outgoing and incoming messages between multiple containers within a single process.

2.3.2 The RPC layer

Remote procedure calls let you, as the name suggest, call functions or methods of remote objects via a network connection (nearly) like you would call local functions. This often leads to more readable code compared to using the lower level *channels*.

Basics

The basic idea behind RPC is as follows: You have a remote object with some methods. On the local side of the connection you have a proxy object which has the same signature, but when you call one of the proxy's methods, it actually sends a message *(method_name, args, kwargs)* to the peer. The peer has a router that maps *method_name* to an actual method. It calls the method and sends its return value back to the proxy. The proxy method returns this value as if it was calculated locally. This works very similarly to how web-frameworks like Django resolve URLs and map them to views.

The following list briefly explains the most important components of aiomas RPC:

Service side:

- An *RPC server:* It starts a server socket and as a *root* object whose methods can be called by clients.
- An *RPC service* (or a hierarchy of services): RPC services are classes with methods that clients can call. Instead of classes with methods you can also use dicts with normal functions. Services can be nested to created hierarchies.
- *RPC routers:* Routers map function names (or paths) to actual methods. An class with an RPC service automatically creates a new router for each of its instances.
- *Exposed* methods: Methods/functions need to be explicitly exposed via a simple decorator. This is a security and safety measure which makes sure that clients can only call functions they are intended to.

Client side:

- An *RPC client:* It represents a network connection to an RPC server and provides a proxy object to its service.
- *RPC proxies:* Proxy objects represent the remote services. They resemble the signature of the services they represent and delegate method calls to them.

Here is a simple example that demonstrate how these components work together:

```
>>> import aiomas
>>>
>>>
>>> class MathServer:
...     # The "Service" creates a router for each instance of "MathServer":
...     router = aiomas.rpc.Service()
...
...     # Exposed methods can be called by clients:
...     @aiomas.expose
...     def add(self, a, b):
...         return a + b
...
>>>
>>> async def client():
...     """Client coroutine: Call the server's "add()" method."""
...     # Connect to the RPC server and get an "RpcClient":
...     rpc_con = await aiomas.rpc.open_connection(('localhost', 5555))
...     # "remote" is a Proxy to the remote service.
...     # We cann call its "add()" method:
...     rep = await rpc_con.remote.add(3, 4)
...     print('What’s 3 + 4?', rep)
...     await rpc_con.close()
>>>
>>> # Start the RPC server with an instance of the "MathServer" service:
>>> server = aiomas.run(aiomas.rpc.start_server(('localhost', 5555), MathServer()))
>>>
>>> aiomas.run(client())
What’s 3 + 4? 7
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

Let’s discuss the details of what we just did:

The class `MathServer` is going to be the root node of our RPC server. Therefore, it needs to be marked as RPC service by giving it a *router* attribute with an `aiomas.rpc.Service` instance. `Service` is a descriptor; when you access the *router* attribute through the `MathServer` class, you get the `Service` instance, but when you access it via a `MathServer` instance, you get an `aiomas.rpc.Router` instance instead. The `Service` descriptor makes sure that every instance of `MathServer` automatically gets its own `Router` instance.

The `add()` method is decorated with `expose()` which makes it available for RPC calls. The arguments and return values of exposed functions must be serializable by the `Codec` used. Numbers, booleans, strings, lists and dicts should always work.

When we start our RPC server (via `aiomas.rpc.start_server()`) we need to pass an instance of our `MathServer` class to it.

In the client, we create an RPC connection via `aiomas.rpc.open_connection()`. It returns an `aiomas.rpc.RpcClient` instance. We can get the proxy to the RPC root node via its `remote` attribute. In contrast to normal method calls, we need to use the `await` (or `yield from`) statement for remote method calls.

Using dictionaries with functions as RPC services

Sometimes, you don’t want or don’t need classes but plain Python functions. With aiomas you can put them in a dict and expose them as an RPC service, too. Here’s a rewrite of out math server example that we discussed in the last section:

```
>>> @aiomas.expose
... def add(a, b):
...     return a + b
...
>>> math_service = aiomas.rpc.ServiceDict({
...     'add': add,
... })
>>>
>>> # Start the RPC server with the math service:
>>> server = aiomas.run(aiomas.rpc.start_server(('localhost', 5555), math_service))
>>>
>>> # The client stays the same as in our last example:
>>> aiomas.run(client())
What's 3 + 4? 7
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

You just need a dict mapping names to the respective functions and wrap it with *aiomas.rpc.ServiceDict*. You can then uses this to start an RPC server.

How to build hierarchies of RPC services

When you want to expose a lot of functions, you may wish to group and categorize them. You can do this by building hierarchies of RPC services (just think of the RPC services as folders and the exposed methods as files, for example). On the client side, you use the `.` operator to access a sub-service (e.g., `root_service.sub_service.method()`).

When you build service hierarchies, you can freely mix class-based and dictionary-based services.

If the parent service is a dictionary, you can add sub services as a new `name: service_instance` pair:

```
>>> @aiomas.expose
... def add(a, b):
...     return a + b
...
>>> # A Sub-service for addition
>>> adding_service = aiomas.rpc.ServiceDict({
...     'add': add,
... })
>>>
>>> # A Sub-service for subtraction
>>> class SubService:
...     router = aiomas.rpc.Service()
...
...     @aiomas.expose
...     def sub(self, a, b):
...         return a - b
...
>>> # Service dict with two sub-services:
>>> root_service = aiomas.rpc.ServiceDict({
...     'addition': adding_service,    # Service dict
...     'subtraction': SubService(),   # Instance(!) of service class
... })
>>>
```

```
>>> async def client():
...     rpc_con = await aiomas.rpc.open_connection(('localhost', 5555))
...     # Call the addition service:
...     rep = await rpc_con.remote.addition.add(3, 4)
...     print('What’s 3 + 4?', rep)
...     # Call the subtraction service:
...     rep = await rpc_con.remote.subtraction.sub(4, 3)
...     print('What’s 4 - 3?', rep)
...     await rpc_con.close()
>>>
>>> server = aiomas.run(aiomas.rpc.start_server(('localhost', 5555), root_service))
>>>
>>> aiomas.run(client())
What’s 3 + 4? 7
What’s 4 - 3? 1
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

As you can see, this is very straight forward. Like a folder that can contain sub-folders and files, a `ServiceDict` can contain sub-services and exposed functions.

Adding sub-services to a service class looks a little bit more complicated, but basically works the same:

```
>>> @aiomas.expose
... def add(a, b):
...     return a + b
...
>>> # A Sub-service for addition
>>> adding_service = aiomas.rpc.ServiceDict({
...     'add': add,
... })
>>>
>>> # A Sub-service for subtraction
>>> class SubService:
...     router = aiomas.rpc.Service()
...
...     @aiomas.expose
...     def sub(self, a, b):
...         return a - b
...
>>> class RootService:
...     # You first have to declare that instances of this class will have
...     # the following sub-services:
...     router = aiomas.rpc.Service(['addition', 'subtraction'])
...
...     def __init__(self):
...         # For each(!) instance, you have to add instances of the
...         # declared sub-services:
...         self.addition = adding_service
...         self.subtraction = SubService()
>>>
>>>
>>> server = aiomas.run(aiomas.rpc.start_server(('localhost', 5555), RootService()))
>>>
>>> # The client remains the same
>>> aiomas.run(client())
What’s 3 + 4? 7
What’s 4 - 3? 1
```

```
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

What makes adding sub-services to classes a bit more complicated is the fact that classes define the service hierarchy but you use instances for the actual RPC servers. That's why you first need to declare at class level which attributes will hold sub-services and then actually add these sub-services in the class' `__init__()`.

You can also manually compose hierarchies with the router's *add()* and *set_sub_router()* methods. These methods give you a bit more flexibility to create service hierarchies on-the-fly:

```
>>> @aiomas.expose
... def add(a, b):
...     return a + b
...
>>> # A Sub-service for addition
>>> adding_service = aiomas.rpc.ServiceDict({
...     'add': add,
... })
>>>
>>> # A Sub-service for subtraction
>>> class SubService:
...     router = aiomas.rpc.Service()
...
...     @aiomas.expose
...     def sub(self, a, b):
...         return a - b
...
>>> class RootService:
...     # In contrast to the last example, we don't declare any sub-services:
...     router = aiomas.rpc.Service()
...
...     def __init__(self):
...         # Add a sub-services via the router's "add()" method:
...         self.addition = adding_service
...         self.router.add('addition')
...
...         # Add a sub-service via the router's "set_sub_router()" method:
...         self.subtraction = SubService()
...         self.router.set_sub_router(self.subtraction.router, 'subtraction')
>>>
>>>
>>> server = aiomas.run(aiomas.rpc.start_server(('localhost', 5555), RootService()))
>>>
>>> # The client remains the same
>>> aiomas.run(client())
What's 3 + 4? 7
What's 4 - 3? 1
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

The method *add()* looks the associated object has an attribute with the specified name that holds the sub services. That service is then exposed via the same name.

Using the method *set_sub_router()*, you can set any router as a sub-router and expose it via the specified name. This provides the most flexibility for building service hierarchies.

Bi-directional RPC: How to allow callbacks from server to client

Aiomas supports bi-directional RPC. That means that not only can a client call server methods, but a server can also call client methods.

For uni-directional RPC, the server specifies an RPC services and a client gets a proxy to it when it makes a connection to the server. For bi-directional RPC, you also need to define a service for your client. The client can pass its service instance as argument of an RPC to the server. The server will then receive a proxy to that service, that it can use to make calls back to the client.

That works because objects with a `router` attribute that is an RPC router can be serialized and be sent to the peer where they get deserialized to an RPC proxy object.

Let’s look at an example to see how it works. The first example uses class-based services:

```
>>> import aiomas
>>>
>>>
>>> class Client:
...     # The client needs to be marked as RPC service:
...     router = aiomas.rpc.Service()
...
...     def __init__(self, name):
...         self.name = name
...
...     async def run(self):
...         # When we open a connection, we need to pass the service instance
...         # ("self" in this case) so that a background task for it can be
...         # started:
...         rpc_con = await aiomas.rpc.open_connection(('localhost', 5555),
...                                                    rpc_service=self)
...
...         # We can now pass the service to the server when we call one of its
...         # methods:
...         rep = await rpc_con.remote.server_method(self)
...         print('Server reply:', rep)
...
...         await rpc_con.close()
...
...     # This method is exposed to the server:
...     @aiomas.expose
...     def get_client_name(self):
...         return self.name
>>>
>>>
>>> class Server:
...     router = aiomas.rpc.Service()
...
...     @aiomas.expose
...     async def server_method(self, client_proxy):
...         # When a client passes a reference to its service, we'll receive it as
...         # a proxy object which we can use to call a client method:
...         client_name = await client_proxy.get_client_name()
...         return 'Client name is "%s"' % client_name
>>>
>>>
>>> server = aiomas.run(aiomas.rpc.start_server(('localhost', 5555), Server()))
>>>
```

```
>>> aiomas.run(Client('Monty').run())
Server reply: Client name is "Monty"
>>>
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

Bi-directional RPC works with class-based as well as dict-based services. Furthermore, if your server or client provide a hierarchy of services, you can not only pass the root service but also any of its sub-services as function arguments.

How to handle remote exceptions

If an RPC raises an error, aiomas wraps it with a *RemoteException* and forwards it to the caller. It also provides you the source (peer name) of the exception and its original traceback:

```
>>> @aiomas.expose
... def fail_badly():
...     raise ValueError('"spam" is not a number')
>>>
>>> service = aiomas.rpc.ServiceDict({'fail_badly': fail_badly})
>>>
>>> async def client():
...     rpc_con = await aiomas.rpc.open_connection(('127.0.0.1', 5555))
...     try:
...         await rpc_con.remote.fail_badly()
...     except aiomas.RemoteException as exc:
...         print('Origin:', exc.origin)
...         print('Traceback:', exc.remote_traceback)
>>>
>>> server = aiomas.run(aiomas.rpc.start_server(('127.0.0.1', 5555), service))
>>>
>>> aiomas.run(client())
Origin: ('127.0.0.1', 5555)
Traceback: Traceback (most recent call last):
  ...
ValueError: "spam" is not a number

>>> server.close()
>>> aiomas.run(server.wait_closed())
```

It is currently not possible to forward the original exception instance, because the caller might not have the required code available (However, I won't rule out the possibility that I might eventually implement this).

How to get a list of connected clients

An RPC service on the server side does not know if or when a new client connects. However, you can pass a *client connected callback* to `aiomas.rpc.start_server()` that cats called once for each new connection. Its only argument is the *RpcClient* for that connection. You can uses this, for example, to close the connection with the client or call the client's exposed methods (if there are some).

```
>>> service = aiomas.rpc.ServiceDict({})
>>>
>>> async def client():
```

```
...     rpc_con = await aiomas.rpc.open_connection(('127.0.0.1', 5555))
...     await rpc_con.close()
>>>
>>> def client_connected_cb(rpc_client):
...     print('Client connected:', rpc_client)
>>>
>>> server = aiomas.run(aiomas.rpc.start_server(('127.0.0.1', 5555), service,
...                                             client_connected_cb))
>>>
>>> aiomas.run(client())
Client connected: <aiomas.rpc.RpcClient object at 0x...>
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

How to handle connection losses

For many reasons, the connection between two endpoints can be lost at any time.

If you are in a coroutine and actively doing RPC, you will get a `ConnectionResetError` thrown into your coroutine if the connection drops:

```
>>> import aiomas
>>>
>>>
>>> async def client():
...     rpc_con = await aiomas.rpc.open_connection(('localhost', 5555))
...     # The server will close the connection when we make the following call:
...     try:
...         await rpc_con.remote.close_connection()
...     except ConnectionResetError:
...         print('Connection lost :(')
>>>
>>>
>>> class Server:
...     router = aiomas.rpc.Service()
...
...     def __init__(self):
...         self.clients = []
...
...     def client_connected(self, client):
...         """*Client connected cb.* that adds new clients to ``self.clients``"""
...         self.clients.append(client)
...
...     @aiomas.expose
...     async def close_connection(self):
...         """Close all open connections and remove them from ``self.clients``."""
...         while self.clients:
...             client = self.clients.pop()
...             await client.close()
>>>
>>> server_service = Server()
>>> server = aiomas.run(aiomas.rpc.start_server(('localhost', 5555),
...                                             server_service,
...                                             server_service.client_connected))
>>>
>>> aiomas.run(client())
```

```
Connection lost :(
>>>
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

If you only serve an RPC service, it gets a little bit more complicated, because RPC services are not connection-aware. However, `aiomas.rpc.RpcClient.on_connection_reset()` lets you register a callback that gets called when the connection is lost. (You get an instance of `RpcClient` as return value from `open_connection()` or via `start_server()`'s *client connected callback*.)

In the following example, the server again has a list of connected clients. But this time, the client disconnects and the server removes the closed connection from its list of clients:

```
>>> import aiomas
>>>
>>>
>>> async def client():
...     rpc_con = await aiomas.rpc.open_connection(('localhost', 5555))
...     await rpc_con.close()
>>>
>>>
>>> class Server:
...     router = aiomas.rpc.Service()
...
...     def __init__(self):
...         self.clients = []
...
...     def client_connected(self, client):
...         # Register a callback that removes the client from our list
...         # when it disconnects:
...         def remove_client(exc):
...             print('Client disconnected :(')
...             self.clients.remove(client)
...
...         client.on_connection_reset(remove_client)
...         print('Client connected :)')
...         self.clients.append(client)
>>>
>>> server_service = Server()
>>> server = aiomas.run(aiomas.rpc.start_server(('localhost', 5555),
...                                              server_service,
...                                              server_service.client_connected))
>>>
>>> aiomas.run(client())
Client connected :)
Client disconnected :(
>>>
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

2.3.3 The channel layer

The channel layer is aiomas' lowest layer of abstraction. It lets you send and receive complete *messages*. In contrast to asyncio's built-in stream protocol which just sends byte strings, messages are JSON-encoded *[0] data (which is a lot more convenient).

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

```
>>> import aiomas
>>>
>>>
>>> async def client():
...     """Client coroutine: Send a greeting to the server and wait for a
...     reply."""
...     channel = await aiomas.channel.open_connection(('localhost', 5555))
...     rep = await channel.send('ohai')
...     print(rep)
...     await channel.close()
>>>
>>>
>>> async def handle_client(channel):
...     """Handle a client connection."""
...     req = await channel.recv()
...     print(req.content)
...     await req.reply('cya')
...     await channel.close()
>>>
>>>
>>> server = aiomas.run(aiomas.channel.start_server(('localhost', 5555), handle_client))
>>> aiomas.run(client())
ohai
cya
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

A communication channel has two sides: The client side is created and returned by `open_connection()`. For each client connection, the server creates a `Channel` instance and starts a new background task of the *client connected callback (client_connected_cb)* to which it passes that channel instance.

Both, the client and server side, can send and receive messages. In the example above, the client starts to send a request and the server side waits for incoming requests. A request has a `content` attribute which holds the actual message. To send a reply, you can either use `Request.reply()` or `Request.fail()`. `Channel.send()` and `Request.reply()` take any data that the channel's codec can serialize (e.g., strings, numbers, lists, dicts, ...). `Request.fail()` takes an exception instance which is raised at the requesting side as `RemoteException`, as the following example demonstrates:

```
>>> import aiomas
>>>
>>>
>>> async def client():
...     """Client coroutine: Send a greeting to the server and wait for a
...     reply."""
...     channel = await aiomas.channel.open_connection(('localhost', 5555))
...     try:
...         rep = await channel.send('ohai')
...         print(rep)
```

[0] Actually, whether JSON is used for encoding, depends on the *codec* that the channel uses. JSON is the default, but you can also use MsgPack or something else. At the bottom of this document, there's a section *explaining aiomas' message format in detail*.

```
...     except aiomas.RemoteException as e:
...         print('Got an error:', str(e))
...     finally:
...         await channel.close()
>>>
>>>
>>> async def handle_client(channel):
...     """Handle a client connection."""
...     req = await channel.recv()
...     print(req.content)
...     await req.fail(ValueError(42))
...     await channel.close()
>>>
>>>
>>> server = aiomas.run(aiomas.channel.start_server(('127.0.0.1', 5555), handle_client))
>>> aiomas.run(client())
ohai
Got an error: Origin: ('127.0.0.1', 5555)
ValueError: 42

>>> server.close()
>>> aiomas.run(server.wait_closed())
```

These are the basics of the channel layer. The following sections answer some detail questions.

How can I use and another codec?

In order to use another codec as the default *JSON* one, just pass the corresponding codec class (e.g., *MsgPack* to *open_connection()* and *start_server()*:

```
>>> import aiomas
>>>
>>> CODEC = aiomas.codecs.MsgPack
>>>
>>> async def client():
...     """Client coroutine: Send a greeting to the server and wait for a
...     reply."""
...     channel = await aiomas.channel.open_connection(('localhost', 5555),
...                                                    codec=CODEC)
...     rep = await channel.send('ohai')
...     print(rep)
...     await channel.close()
>>>
>>>
>>> async def handle_client(channel):
...     """Handle a client connection."""
...     req = await channel.recv()
...     print(req.content)
...     await req.reply('cya')
...     await channel.close()
>>>
>>>
>>> server = aiomas.run(aiomas.channel.start_server(('localhost', 5555), handle_client,
...                                                 codec=CODEC))
>>> aiomas.run(client())
ohai
```

```
cya
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

Note, that the codecs `aiomas.codecs.MsgPack` and `aiomas.codecs.MsgPackBlosc` are not available by default but have to *be explicitly enabled*.

How can I serialize custom data types?

Both, `open_connection()` and `start_server()` take a list of *extra_serializers*. Such a serializer is basically a function returning a three-tuple *(type, serialize, deserialize)*. You can find more details in the *codecs guide*. Here is just a simple example:

```
>>> import aiomas
>>>
>>>
>>> class MyType:
...     """Our serializable type."""
...     def __init__(self, value):
...         self.value = value
...
...     def __repr__(self):
...         return '%s(%r)' % (self.__class__.__name__, self.value)
>>>
>>>
>>> def serialize_mytype(obj):
...     """Return a JSON serializable version "MyType" instances."""
...     return obj.value
>>>
>>>
>>> def deserialize_mytype(value):
...     """Make a "MyType" instance from *value*."""
...     return MyType(value)
>>>
>>>
>>> def mytype_serializer():
...     return (MyType, serialize_mytype, deserialize_mytype)
>>>
>>>
>>> EXTRA_SERIALIZERS = [mytype_serializer]
>>>
>>>
>>> async def client():
...     """Client coroutine: Send a greeting to the server and wait for a
...     reply."""
...     channel = await aiomas.channel.open_connection(
...         ('localhost', 5555), extra_serializers=EXTRA_SERIALIZERS)
...     rep = await channel.send(['ohai', MyType(42)])
...     print(rep)
...     await channel.close()
>>>
>>>
>>> async def handle_client(channel):
...     """Handle a client connection."""
...     req = await channel.recv()
...     print(req.content)
...     await req.reply(MyType('cya'))
...     await channel.close()
>>>
>>>
>>> server = aiomas.run(aiomas.channel.start_server(('localhost', 5555), handle_client,
...                                                  extra_serializers=EXTRA_SERIALIZERS)
>>> aiomas.run(client())
['ohai', MyType(42)]
MyType('cya')
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

A shorter version for common cases is using the `aiomas.codecs.serializable()` decorator:

```
>>> import aiomas
>>>
>>>
>>> @aiomas.codecs.serializable
... class MyType:
...     """Our serializable type."""
...     def __init__(self, value):
...         self.value = value
>>>
>>>
>>> EXTRA_SERIALIZERS = [MyType.__serializer__]
>>>
>>>
>>> async def client():
...     """Client coroutine: Send a greeting to the server and wait for a
...     reply."""
...     channel = await aiomas.channel.open_connection(
...         ('localhost', 5555), extra_serializers=EXTRA_SERIALIZERS)
...     rep = await channel.send(['ohai', MyType(42)])
...     print(rep)
...     await channel.close()
>>>
>>>
>>> async def handle_client(channel):
...     """Handle a client connection."""
...     req = await channel.recv()
...     print(req.content)
...     await req.reply(MyType('cya'))
...     await channel.close()
>>>
>>>
>>> server = aiomas.run(aiomas.channel.start_server(('localhost', 5555), handle_client,
...                                                  extra_serializers=EXTRA_SERIALIZERS)
>>> aiomas.run(client())
['ohai', MyType(value=42)]
MyType(value='cya')
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

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 errno
>>> 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 != errno.EADDRINUSE:
...                 # 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())
```

Connection timeouts / Starting clients before the server

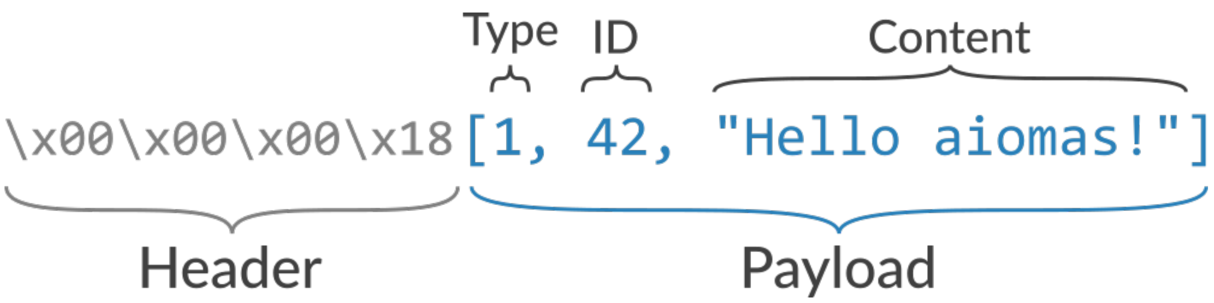
Sometimes, you need to start a client before the server is started. Therefore, the function `open_connection()` lets you specify a timeout. It repeatedly retries to connect until *timeout* seconds have passed. By default, *timeout* is 0 which means there is only one try.

```
>>> import asyncio
>>> import aiomas
>>>
>>>
>>> async def client():
...     """Client coroutine: Send a greeting to the server and wait for a
...     reply."""
...     # Try to connect for 1s:
...     channel = await aiomas.channel.open_connection(('localhost', 5555),
...                                                    timeout=1)
...     rep = await channel.send('ohai')
...     print(rep)
...     await channel.close()
>>>
>>>
>>> async def handle_client(channel):
...     """Handle a client connection."""
...     req = await channel.recv()
...     print(req.content)
...     await req.reply('cya')
...     await channel.close()
>>>
>>>
>>> # Start the client in background, ...
>>> t_client = asyncio.async(client())
>>> # wait 0.5 seconds, ...
>>> aiomas.run(asyncio.sleep(0.5))
>>> # and finally start the server:
>>> server = aiomas.run(aiomas.channel.start_server(('localhost', 5555), handle_client))
>>> aiomas.run(t_client)
ohai
cya
>>> server.close()
>>> aiomas.run(server.wait_closed())
```

How exactly do messages look like?

This section explains how aiomas messages look and how they are constructed. You can easily implement this protocol in other languages, too, and write programs that can communicate with aiomas.

Network messages consists of a four bytes long *header* and a *payload* of arbitrary length. The header is an unsigned integer (uint32) in network byte order (big-endian) and stores the number of bytes in the payload. The payload itself is an encoded †[0] list containing the message type, a message ID and the actual content:



Messages send between two peers must follow the request-reply pattern. That means, every request that one peer makes must be responded to by the other peer. Request use the message type `0`, replies use `1` for success or `2` to indicate a failure. The message ID is an integer that is unique for every request that a network socket makes. Replies (no matter if successful or failed) need to use the message ID of the corresponding request.

On the channel layer, the *content* of a request can be anything. On the RPC level, it a three-tuple *(function_path, args, kwargs)*, e.g.:

```
[function, [arg0, arg1, ...], {kwarg0: val0, kwarg1: val1}]
```

Thereby, *function* is always a string containing the name of an exposed functions; if you use nested services, sub-services and the function names are separated by slashes (`/`) as in URLs. The type of the arguments and keyword arguments may vary depending on the function.

The content types of replies are the same for both, the channel layer and the RPC layer. Normal (successful) replies can be anything. The content of failure replies are strings with the error message and/or a stack trace.

Note: If the *JSON* codec is used, aiomas messages are compatible with simpy.io (and therewith with the co-simulation framework mosaik, too).

[0] Depending on the *codec* you use, the payload may be a UTF-8 encoded JSON string (`json.dumps().encode('utf-8')`) (this is the default), a MsgPack list (`msgpack.packb()`), or whatever else the codec produces.

2.3.4 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.

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`.

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.

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 *Bi-directional RPC: How to allow callbacks from server to client* for details.

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

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:

```python
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])
```

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.

2.3.5 Container clocks

Clocks and time are a very important instrument and required, if your agents want to delay the execution of a task for some time, schedule a task at a certain time or just need to define a timeout.

Usually, the real (or wall-clock) time is used for this. In some contexts, however, you need a different notion of time – for example if you want to couple a multi-agent system with external simulators that usually run faster than real-time.

For this reason, every agent container provides a clock via its `clock` attribute. The default clock is the real-time clock that asyncio uses (`AsyncioClock`).

An alternative clock is the `ExternalClock`. The time of this clock can be set by external processes so that the time within your agent system passes as fast (or slow) as in that external process.

The benefit of using aiomas' clocks compared to just using what asyncio offers is, that you can easily switch clocks (e.g., from the `AsyncioClock` to the `ExternalClock`) without touching the agents:

```
>>> import aiomas
>>>
>>>
>>> CLOCK = aiomas.AsyncioClock()
>>> # CLOCK = aiomas.ExternalClock('2016-01-01T00:00:00')
>>>
>>> class Sleeper(aiomas.Agent):
...     async def run(self):
...         # await asyncio.sleep(.5)  # <-- Don't use this!
...         # Depending on the clock used, this sleeps for a "real" half
...         # second or whatever the ExternalClock tells you:
...         await self.container.clock.sleep(.5)
>>>
>>> container = aiomas.Container.create(('127.0.0.1', 5555), clock=CLOCK)
>>> agent = Sleeper(container)
>>> aiomas.run(agent.run())
>>> container.shutdown()
```

(If you uncomment the ExternalClock in the example above, your program won't terminate because there's no process that sets its time.)

Date/time representations

All clocks represent time as a monotonically increasing number (not necessarily with a defined initial value) and as date/time object (for which the arrow package is used).

You can get the numeric time via the clock's `time()` method. Its usage is comparable to that of Python's `time.monotonic()` function.

The method `utcnow()` returns an `Arrow` object with the current date and time in UTC.

Note: You should work with UTC dates as much as possible. Input dates with a local timezone should be converted to UTC as early as possible. If you output dates, convert them as late as possible back to local time.

Doing date and time calclulations in UTC saves you from a lot of bugs, i.e., when dealing with daylight-saving times.

This blog post by Armin Ronacher and this talk by Taavi Burns provide more background to the issue.

Sleeping

The container clock provides tasks that let your agent sleep *for* a given amount of time or *until* a given time is reached.

In order to sleep for a given time, you have to use the method `sleep()` with the number of seconds (as float) that you want to sleep.

The method `sleep_until()` also accepts a number in seconds (which must be greater than the current value of `time()`) or an `Arrow` date object (which must be greater than the current value of `utcnow()`).

Both methods return a future which you have to `await` / `yield from` in order to actually sleep.

Scheduling tasks

Comparably to sleeping, you can schedule the future execution of a task *in* a given period of time or *at* a given time.

The method `call_in()` will run the specified task after a delay *dt* in seconds; `BaseClock.call_at()` will run the task at the specified date (either in seconds or as `Arrow` date). You can only pass positional arguments to these methods, because that's what the underlying *asyncio* functions allow.

Both methods are normal functions that return a handle to the scheduled call. You can use this handle to `cancel()` the scheduled execution of the task.

How to use the ExternalClock

Remember the first example which did not actually work if you used the `ExternalClock`? Here is a fully working version of it:

```
>>> import asyncio
>>> import time
>>>
>>> import aiomas
>>>
>>>
>>> CLOCK = aiomas.ExternalClock('2016-01-01T00:00:00')
>>>
>>> class Sleeper(aiomas.Agent):
...     async def run(self):
...         print('Gonna sleep for 1s ...')
...             await self.container.clock.sleep(1)
>>>
>>>
>>> async def clock_setter(factor=0.5):
...     """Let the time pass *factor* as fast as real-time."""
...     while True:
...         await asyncio.sleep(factor)
...         CLOCK.set_time(CLOCK.time() + 1)
>>>
>>> container = aiomas.Container.create(('127.0.0.1', 5555), clock=CLOCK)
>>>
>>> # Start the process that sets the clock:
>>> t_clock_setter = asyncio.async(clock_setter())
>>>
>>> # Start the agent an measure how long he runs in real-time:
>>> agent = Sleeper(container)
>>> start = time.monotonic()
>>> aiomas.run(agent.run())
Gonna sleep for 1s ...
>>> print('Agent process finished after %.1fs' % (time.monotonic() - start))
Agent process finished after 0.5s
>>>
>>> _ = t_clock_setter.cancel()
>>> container.shutdown()
```

Now that we have a background process that steps the time forward, the example actually terminates.

In scenarios where you want to couple you agent system with the clock of another system, the `clock_setter()` process would not sleep but receive clock updates from that other process and use these updates to set the agent's clock to a new time.

If you distribute your agent system over multiple processes, make sure that you spread the clock updates to all agent containers. Therefore, the *`Manager`* agent in the *`aiomas.subproc`* exposes a *`set_time()`* method that an agent in your master process can call.

2.3.6 Testing and debugging

Here are some general rules and ideas for developing and debugging distributed systems with aiomas:

- Distributed systems are complex. Always start as simple as possible. Examine and understand the behavior of that system. Start adding a bit more complexity. Repeat.
- I find using a debugger does not work very well with async., distributed systems, so I tend to add a lot of logging and or `print()`s to my code for debugging purposes.
- Read Develop with asyncio.
- If you enable asyncio's debug mode, aiomas also falls into debug mode. It gives you better / more detailed exceptions in some cases. This impacts performance, so it isn't activated always.
- Write unit and integration tests and run them as often as possible. Also check that your tests will fail if they should.

Testing coroutines and agents with pytest

My preferred testing tool is pytest. The plug-in pytest-asyncio makes testing asyncio based programs a lot easier.

As an introduction, I also suggest reading my articles on testing with asyncio. They are especially helpful if you are using the channel and RPC layers. Testing agent systems is a bit “easier” (in the sense that the tests are easier to setup). You can, of course, also look at aiomas’ test suite itself.

Here is a small example that demonstrate how you could test an agent. In this case, the agent class itself and the tests for it are in the same module. In real life, you would have the agent and its test in separate packages (e.g., `exampleagent.py` and `test_exampleagent.py`).

```
import pytest
import aiomas

#
# Production code (exampleagent.py)
#


class ExampleAgent(aiomas.Agent):
    async def run(self, target_addr, num):
        remote_agent = await self.container.connect(target_addr)
        return (await remote_agent.service(num))

    @aiomas.expose
    async def service(self, val):
        await self.container.clock.sleep(0.001)
        return val


#
# Testing code (test_exampleagent.py)
#


@pytest.yield_fixture
def container(event_loop, unused_tcp_port):
    """This fixture creates a new Container instance for every test and binds
    it to a random port.

    It requires the *event_loop" fixture, so every test will also have a fresh
    event loop.

    """
    # Create container and bind its server socket to a random port:
    c = aiomas.Container.create(('127.0.0.1', unused_tcp_port))

    # Yield the container to the test case:
    yield c

    # Clean-up that is run after the test finished:
    c.shutdown()


# The "@pytest.mark.asyncio" decorator allows you do use "await"/"yield from"
# directly within your test case.
```

```
#
# The "container" argument tells pytest to pass the return/yield value of the
# corresponding fixture to your test.
@pytest.mark.asyncio
async def test_example_agent(container):
    num = 42
    # Start two agents:
    agents = [ExampleAgent(container) for _ in range(2)]
    # Run the 1st one and let it connect to the 2nd one.  Check the return
    # value of the 1st one's run() task:
    res = await agents[0].run(agents[1].addr, num)
    assert res == num
```

2.3.7 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).

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.

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.

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.

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)
...     await 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')
...     await 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())
```

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()
```

2.4 API reference

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

2.4.1 `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
get_queue(queue_id)	Return a *LocalQueue* instance for the given *queue_id*.
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
run([until])	Run the event loop forever or until the task/future *until* is finish

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.
AsyncioClock()	`asyncio` based real-time clock.
Container(base_url, clock, connect_kwargs)	Container for agents.
ExternalClock(utc_start[, init_time])	A clock that can be set by external process in order to synchronize it
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*
SSLCerts(cafile, certfile, keyfile)	`namedtuple()` storing the names of a CA file, a

2.4.2 `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 for a TCP socket, a path for a Unix domain socket, or a *LocalQueue* instance as returned by the `aiomas.local_queue.get_queue()` function.

  TCP sockets

  If `host` is `'0.0.0.0'` or `'::'`, 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.

  LocalQueue

  In contrast to TCP, multiple LocalQueue connections between containers (within the same thread and OS process) send and receive message in a deterministic order, which is useful for testing and debugging.

  LocalQueue instances should be retrieved via the `aiomas.local_queue.get_queue()` function (which also available as `aiomas.get_queue()`). This function always returns the same instance for a given queue ID.

- **`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*, *timeout=0*)

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'`).

With a *timeout* of 0 (the default), there will only be one connection attempt before an error is raised (`ConnectionRefusedError` for TCP sockets and LocalQueue, `FileNotFoundError` for Unix domain sockets). If you set *timeout* to a number > 0 or `None`, this function will try to connect repeatedly for at most that many seconds (or indefinitely) before an error is raised. Use this if the remote agent's container may not yet exist.

This function is a coroutine.

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.

Set *async* to `True` if the event loop is already running (e.g., because you are in a coroutine). 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.

2.4.3 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=()*, *timeout=0*, ***kwds*)
Return a *Channel* connected to *addr*.

This is a convenience wrapper for `asyncio.BaseEventLoop.create_connection()`, `asyncio.BaseEventLoop.create_unix_connection()`, and *aiomas.local_queue.create_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. If *addr* is a *aiomas.local_queue* instance, a *LocalQueue* connection will be created.

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()*.

With a *timeout* of 0 (the default), there will only be one connection attempt before an error is raised (`ConnectionRefusedError` for TCP sockets and LocalQueue, `FileNotFoundError` for Unix domain sockets). If you set *timeout* to a number > 0 or `None`, this function will try to connect repeatedly for at most that many seconds (or indefinitely) before an error is raised. Use this if you need to start the client before the server.

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

This function is a coroutine.

`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()`, `asyncio.BaseEventLoop.create_unix_server()`, and *`aiomas.local_queue.create_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. If *addr* is a *`aiomas.local_queue`* instance, a *LocalQueue* server 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.

This function is a coroutine.

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(*len_bytes*, *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.

    This method is a coroutine.

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

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

    This method is a coroutine.

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:

- ValueError if the message is too long (the length of the encoded message does not fit into a *long*, which is ~ 4 GiB).
- *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.

This method is a coroutine.

close()

Coroutine that closes the channel and waits for all sub tasks to finish.

get_extra_info(*name*, *default=None*)

Wrapper for `asyncio.BaseTransport.get_extra_info()`.

2.4.4 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.

2.4.5 `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.

2.4.6 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.

    origin = None
        Peername (producer of the exception)

    remote_traceback = None
        Original traceback

exception aiomas.exceptions.**SerializationError**
    Raised when an object cannot be serialized.

2.4.7 aiomas.local_queue

The local queue transport roughly mimics a normal TCP transport, but it sends and receives messages via two `asyncio.Queue` instances.

Its purpose is to aid the development and debugging of complex networking algorithms and distributed or multi-agent systems. In contrast to normal network transports, messages send via the *LocalQueueTransport* will always arrive in a deterministic order [1].

This transport does *not* work across multiple processes and is *not* thread safe, so it should only be used within a single thread and process.

The easiest way to use it is to create a *LocalQueue* instance via the *get_queue()* function and pass it to *aiomas.channel.start_server()*/*aiomas.channel.open_connection()* or *aiomas.agent.Container.create()* as *addr* argument.

[1] Actually, message sent via a single TCP connection also arrive at a deterministic order (this is a property of the TCP/IP protocol). So the LocalQueue transport won't give you any benefits in this case.

However, if you have multiple connections to the same server and send message through them in parallel, it's no longer deterministic in which order the messages arrive from the different connections. In this case, the LocalQueue transport can help you.

`aiomas.local_queue.`**`get_queue`**`(`*queue_id*`)`

Return a `LocalQueue` instance for the given *queue_id*.

If no instance is cached yet, create a new one.

Queue IDs must be strings and must not contain the `/` character. Raise a `ValueError` if these rules are violated.

`aiomas.local_queue.`**`clear_queue_cache`**`()`

Clear the global queue cache.

`aiomas.local_queue.`**`create_connection`**`(`*protocol_factory*, *lq*, ***, *loop=None*, ***kwds*`)`

Connect to a `LocalQueue` *lq*.

The *protocol_factory* must be a callable returning a protocol instance.

Before a connection to *lq* can be made, a server must be started for this instance (see `create_server()`).

This function is a coroutine which will try to establish the connection in the background. When successful, the coroutine returns a (transport, protocol) pair.

`aiomas.local_queue.`**`create_server`**`(`*protocol_factory*, *lq*, ***kwds*`)`

Create a `LocalQueue` server bound to *lq*.

The *protocol_factory* must be a callable returning a protocol instance.

Return a `LocalQueueServer` instance. That instance is also set as `server` for *lq*.

This function is a coroutine.

class `aiomas.local_queue.`**`LocalQueue`**`(`*queue_id*`)`

An instance of this class serves as transport description when creating a server or connection.

The functions `create_server()` and `create_connection()` both require an instance of this class. Alternatively, instances of this class can be passed as *addr* argument to `aiomas.channel.start_server()` and `aiomas.channel.open_connection()`

A server needs to be started before any connections can be made.

`queue_id`

The queue's ID.

`server`

The `LocalQueueServer` instance that was bound to this instance or `None` if no server has yet been started.

`set_server``(`*server*`)`

Set a `LocalQueueServer` instance.

Raise a `RuntimeError` if a server has already been bound to this instance.

This method is called by `create_server()`.

`unset_server``()`

Unset the server from this instance.

This method is called when the server is closed (see `LocalQueueServer.close()`).

`new_connection``(`*sendq*, *recvq*`)`

Create a connection endpoint on the server side.

This method is called by `create_connection()`.

sendq and *recvq* are the queues used for sending and receiving messages to and from the client.

class `aiomas.local_queue.`**`LocalQueueServer`**(*protocol_factory*, *lq*)

Implements `asyncio.events.AbstractServer`. An instance of this class is returned by `create_server()`.

lq is the `LocalQueue` instance that this server was bound to.

protocol_factory is a callable that is called for each new client connection in order to create a new protocol instance.

`lq`

The `LocalQueue` the server is bound to.

`new_connection`(*sendq*, *recvq*)

Create a new protocol and transport instance.

Call the *protocol factory*, create a new `LocalQueueTransport` with *sendq* and *recvq* and wire them together.

Called by `create_connection()` via `LocalQueue.new_connection()`.

`close`()

Close the server and unset this instance from the associated `LocalQueue` instance.

`wait_closed`()

Immediately return (there's nothing to wait for).

This method is a coroutine.

class `aiomas.local_queue.`**`LocalQueueTransport`**(*lq*, *sendq*, *recvq*, *protocol*)

Implements `asyncio.transports.Transport`.

A *LocalQueueTransport* has to asynchronous queues (instances of `asyncio.Queue`) – one for sending messages to the other side and one for receiving messages from it.

`close`()

Close the transport.

Buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's `connection_lost()` method will (eventually) be called with `None` as its argument.

`write`(*data*)

Write some data bytes to the transport.

This does not block; it buffers the data and arranges for it to be sent out asynchronously.

`can_write_eof`()

Return `False`. This transport does not support `write_eof()`.

`abort`()

Close the transport immediately.

Buffered data will be lost. No more data will be received. The protocol's `connection_lost()` method will (eventually) be called with `None` as its argument.

2.4.8 `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 peer via `Proxy` instances. Optionally, they can provide their own RPC service so that the peer can make calls as well.

An RPC service is an object with a `router` attribute which is an instance of `Router`. A router resolves paths requested by the peer. 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 an be attached to both, classes and dictionaries with functions. Dictionaires need to be wrapped with a `ServiceDict`. Classes need to have a `Service` class attribute named `router`. `Service` is a descriptor which creates a `Router` for every instance of that class.

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*, **, *rpc_service=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 *rpc_service* to allow the peer to call back to us.

This function is a coroutine.

`aiomas.rpc.`**`start_server`**`(`*addr*, *rpc_service*, *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.

rpc_service must be an RPC service (an object with a `router` attribute that is an instance of `Router`).

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.

This function is a coroutine.

`aiomas.rpc.`**`rpc_service_process`**`(`*rpc_client*, *router*, *channel*`)`

RPC service process for a connection *rpc_lient*.

Serves 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.

This function is a coroutine.

`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.`**`ServiceDict`**(*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.`**`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.`**`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.`**RpcClient**(*channel, rpc_service=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 *rpc_service* is not `None`, it will also start its own RPC service so the peer can call the functions we provide.

channel
The communication `Channel` of this instance.

service
The RPC service process for this connection.

remote
A `Proxy` for remote methods.

on_connection_reset(*callback*)
Add a *callback* that gets called if the peer closes the connection and thus causing the `service` process to abort.

callback is a callable with a single argument, the exception that the `service` process raises if the connection is reset by the peer.

If this method is called multiple times, override the current callback with the new one. If *callback* is `None`, delete the current callback.

Raise a `ValueError` if *callback* is neither callable nor `None`.

Raise a `RuntimeError` if this instance has not service task running.

close()
Coroutine that closes the connection and waits for all sub tasks to finish.

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).

2.4.9 aiomas.subproc

This module helps you to *start()* an agent container in a new subprocess. The new container will have a *Manager* agent that allows the master process to spawn other agents in the new container.

The following example demonstrate how you can build a nice CLI with the click around this module. The script will start you a container with an *ExternalClock* and the *MsgPackBlosc* codec:

```
# container.py
import logging

import aiomas
import arrow
import click


def validate_addr(ctx, param, value):
    try:
        host, port = value.rsplit(':', 1)
        return (host, int(port))
    except ValueError as e:
        raise click.BadParameter(e)


def validate_start_date(ctx, param, value):
    try:
        arrow.get(value)  # Check if the date can be parsed
    except arrow.parser.ParserError as e:
        raise click.BadParameter(e)
    return value

@click.command()
@click.option('--start-date', required=True,
              callback=validate_start_date,
              help='Start date for the agents (ISO-8601 compliant, e.g.: '
                   '2010-03-27T00:00:00+01:00')
@click.option('--log-level', '-l', default='info', show_default=True,
              type=click.Choice(['debug', 'info', 'warning', 'error',
                                 'critical']),
              help='Log level for the MAS')
@click.argument('addr', metavar='HOST:PORT', callback=validate_addr)
def main(addr, start_date, log_level):
    logging.basicConfig(level=getattr(logging, log_level.upper()))
    clock = aiomas.ExternalClock(start_date, init_time=-1)
    codec = aiomas.codecs.MsgPackBlosc
    task = aiomas.subproc.start(addr, clock=clock, codec=codec)
    aiomas.run(until=task)


if __name__ == '__main__':
    main()
```

Example usage: **python container.py --start-date=2010-03-27T00:00:00+01:00 localhost:5556**.

Note: You should use `sys.executable` instead of just `'python'` when you start a new subprocess from within a Python script to make sure you use the correct (same) interpreter.

`aiomas.subproc.`**`start`**`(`*addr*, ***container_kwargs*`)`

Coroutine that starts a container with a `Manager` agent.

The agent will connect to *addr* `('host', port)` and wait for commands to spawn new agents within its container.

The *container_kwargs* will be passed to `aiomas.agent.Container.create()` factory function.

This coroutine finishes after `Manager.stop()` was called or when a `KeyboardInterrupt` is raised.

class `aiomas.subproc.`**`Manager`**`(`*container*`)`

An agent that can start other agents within its container.

If the container uses an `ExternalClock`, it can also set the time for the container's clock.

`spawn``(`*qualname*, **args*, ***kwargs*`)`

Create a new instance of an agent and return a proxy to it and its address.

qualname is a string defining a class (or factory method/coroutine) for instantiating the agent (see `aiomas.util.obj_from_str()` for details). *args* and *kwargs* get passed to this callable as positional and keyword arguemnts respectively.

This is an exposed coroutine.

`set_time``(`*time*`)`

Set the agent's container's time to *time*.

This only works if the container uses an `ExternalClock`.

This is an exposed function.

`stop``()`

Triggers the *stop_received* future of this agent causing its container process to shutodwn and terminate.

This is an exposed function.

2.4.10 `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.

`aiomas.util.`**`obj_from_str`**`(`*obj_path*`)`

Return the object that the string *obj_path* points to.

The format of *obj_path* is `mod:obj` where *mod* is a (possibly nested) module name and *obj* is an `.` separate object path, for example:

```
module:Class
module:Class.function
package.module:Class
package.module:Class.function
```

Raise a `ValueError` if the *obj_path* is malformed, an `ImportError` if the module cannot be imported or an `AttributeError` if an object does not exist.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index