Communicating with other agents

Introduction

Communication is an essential feature agent network. It allows agents to work together, exchange information, and forms an organic marketplace.

In this guide, we will explore two methods of communication between agents:

- Local communication
- .
- Remote communication
- via theAlmanac Contract ↗
- .

Let's start withlocal communication . This is the first step you would need to undertake to familiarize yourself with the code syntax we will be using in theremote communication section.

ℹ Local communication is important for debugging purposes.

Agents: Local Communication

Walk-through

The first step to better understand how agents communicate is to introduce how 2 agents perform a local communication. Let's consider a basic example in which two agents say hello to each other.

1. First of all, let's create a Python script for this task and name it by running:touch agents_communication.py
2. Then, we importAgent
3. ,Context
4. ,Bureau
5. , andModel
6. from the uagents library and we then define the message structure for messages to be exchanged between the agents using the classModel
7. :
8. from
9. uagents
10. import
11. Agent
12. ,
13. Bureau
14. ,
15. Context
16. ,
17. Model
18. class
19. Message
20. (
21. Model
22.):
23. message
24. :
25. str
26. TheMessage
27. class defines the structure of message we can receive. In this example it's just a string, but it could be a simple integer, or a complex object too.
28. Now we create two agent instances,sigmar
29. andslaanesh
30. , withname
31. andseed
32. parameters:
33. sigmar
34. =
35. Agent
36. (name
37. =
38. "sigmar"

, seed
=
"sigmar recovery phrase"
)
slaanesh
=
Agent
(name
=
"slaanesh"
, seed
=
"slaanesh recovery phrase"
)
In this example we're running multiple agents from one file.
Let's now definesigmar
's behaviors. We need to define a function forsigmar
to send messages toslaanesh
periodically:
@sigmar
.
on_interval
(period
=
3.0
)
async
def
send_message
(
ctx
:
Context):
await
ctx
.
send
(slaanesh.address,
Message
(message
=
"hello there slaanesh"
))
We can use the.on_interval()
decorator to define a coroutinesend_message()
function that will be called every 3 seconds. The coroutine function sends a message toslaanesh
using thectx.send()
method of theContext
object.
We then need to define asigmar_message_handler()
function forsigmar
to manage incoming messages:
@sigmar
.
on_message
(model
=
Message)
async
def
sigmar_message_handler
(
ctx
:
Context
,
sender
:

str
,
msg
:
Message):
ctx
.
logger
.
info
(
f
"Received message from
{
sender
}
:
{
msg.message
}
"
)
This defines the coroutine functionsigmar_message_handler()
that serves as a message handler forsigmar
. It is triggered wheneversigmar
receives a message of typeMessage
.
Let's now define the behavior of our second agent,slaanesh
:
@slaanesh
.
on_message
(model
=
Message)
async
def
slaanesh_message_handler
(
ctx
:
Context
,
sender
:
str
,
msg
:
Message):
ctx
.
logger
.
info
(
f
"Received message from
{
sender
}
:
{
msg.message
}
"
)
await

175. ctx
176. .
177. send
178. (sigmar.address,
179. Message
180. (message
181. =
182. "hello there sigmar"
183.))
184. Same assigmar
185. however, we makeslaanesh
186. compose a response message to be sent back using thectx.send()
187. method withsigmar.address
188. as the recipient address and an instance of theMessage
189. model as the message payload.
190. It would also be valid to respond to the sender:
191. await
192. ctx
193. .
194. send
195. (sender.address,
196. Message
197. (message
198. =
199. "hello there sigmar"
200.))
201. Let's then use theBureau
202. class to create aBureau
203. object. This will allow us to run agents together in the same script:
204. bureau
205. =
206. Bureau
207. ()
208. bureau
209. .
210. add
211. (sigmar)
212. bureau
213. .
214. add
215. (slaanesh)
216. if
217. **name**
218. ==
219. "**main**"
220. :
221. bureau
222. .
223. run
224. ()
225. Save the script.

The complete script should be looking as follows:

agents_communication.py from uagents import Agent , Bureau , Context , Model

class

Message (Model): message :

str

sigmar

Agent (name = "sigmar" , seed = "sigmar recovery phrase") slaanesh =

Agent (name = "slaanesh" , seed = "slaanesh recovery phrase")

@sigmar . on_interval (period = 3.0) async

def

send_message (ctx : Context): await ctx . send (slaanesh.address, Message (message = "hello there slaanesh"))

@sigmar . on_message (model = Message) async

def

sigmar_message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ")

@slaanesh . on_message (model = Message) async

def

slaanesh_message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ") await ctx . send (sigmar.address, Message (message = "hello there sigmar"))

bureau

Bureau () bureau . add (sigmar) bureau . add (slaanesh)

if

name

==

"**main**" : bureau . run () We are now ready to run the script:python agents_communication.py

The output would be:

[sigmar]: Received message from agent1q0mau8vkmg78xx0sh8cyl4tpl4ktx94pqp2e94cylu6haugt2hd7j9vequ7: hello there sigmar [slaanesh]: Received message from agent1qww3ju3h6kfcuqf54gkghvt2pqe8qp97a7nzm2vp8plfxflc0epzcjsv79t: hello there slaanesh [sigmar]: Received message from agent1q0mau8vkmg78xx0sh8cyl4tpl4ktx94pqp2e94cylu6haugt2hd7j9vequ7: hello there sigmar [slaanesh]: Received message from agent1qww3ju3h6kfcuqf54gkghvt2pqe8qp97a7nzm2vp8plfxflc0epzcjsv79t: hello there slaanesh [sigmar]: Received message from agent1q0mau8vkmg78xx0sh8cyl4tpl4ktx94pqp2e94cylu6haugt2hd7j9vequ7: hello there sigmar

Agents Remote Communication: the Almanac Contract

To speak, search or be found, your agent must register to the Almanac contract ↗ . Agents then query this to retrieve an HTTP endpoint for a recipient agent. Registration in the Almanac ↗ requires paying a small fee, so make sure to have enough funds to allow for this. You can query the Almanac now, by using the search feature on Agentverse ↗(opens in a new tab) .

Whenever an agent registers in the Almanac, it must specify the service endpoints ↗ alongside a weight parameter for each endpoint provided. Agents trying to communicate with your agent, will choose the service endpoints using a weighted random selection.

Here, we show you how to create two agents and make them remotely communicate by registering and using the Almanac Contract.

Walk-through

The first step would be to create two different Python scripts for this task, each one representing a remote agent:

Slaanesh:touch remote_agents_slaanesh.py

Sigmar:touch remote_agents_sigmar.py

Let's start by defining the script forsigmar .

Sigmar

1. Inremote_agents_sigmar.py
2. script, we would need to import the necessary classes from theuagents
3. (Agent
4. ,Context
5. , andModel
6.) and fromuagents.setup
7. (fund_agent_if_low
8.). We then need to define the message structure for messages to be exchanged between agents using the classModel
9. , as well as theRECIPIENT_ADDRESS
10. (slaanesh's address). Note that if you don't know slaanesh's address yet, you can useprint(slaanesh.address)
11. after defining agentslaanesh
12. to get this information. This is the address towards whichsigmar
13. will send messages:
14. from
15. uagents
16. import
17. Agent
18. ,
19. Context
20. ,
21. Model
22. from
23. uagents
24. .
25. setup
26. import
27. fund_agent_if_low
28. class
29. Message
30. (
31. Model
32.):
33. message
34. :
35. str
36. RECIPIENT_ADDRESS
37. =
38. "agent1q2kxet3vh0scsf0sm7y2erzz33cve6tv5uk63x64upw5g68kr0chkv7hw50"
39. Let's now create our agent,sigmar
40. , by providingname
41. ,seed
42. ,port
43. , andendpoint
44. . Also, make sure it has enough funds to register in the Almanac contract:
45. sigmar
46. =
47. Agent
48. (
49. name
50. =
51. "sigmar"
52. ,
53. port
54. =
55. 8000
56. ,
57. seed
58. =
59. "sigmar secret phrase"
60. ,
61. endpoint

=
[
"http://127.0.0.1:8000/submit"
],
)
fund_agent_if_low
(sigmar.wallet.
address
())
On the Fetch.ai testnet, you can use thefund_agent_if_low
function. This checks if the balance of the agent's wallet is below a certain threshold, and if so, sends a transaction to fund the wallet with a specified amount of cryptocurrency. In this case, it checks if the balance ofsigmar
's wallet is low and funds it if necessary.
We are ready to definesigmar
's behaviors. Let's start with a function forsigmar
to send messages:
@sigmar
.
on_interval
(period
=
2.0
)
async
def
send_message
(
ctx
:
Context):
await
ctx
.
send
(RECIPIENT_ADDRESS,
Message
(message
=
"hello there slaanesh"
)
Here, the.on_interval()
decorator schedules thesend_message()
function to be run every 2 seconds. Inside the function, there is an asynchronous call indicated by thectx.send()
method. This call sends a message with the content"hello there slaanesh"
to theRECIPIENT_ADDRESS
.
We then need to define a function forsigmar
to handle incoming messages from other agents:
@sigmar
.
on_message
(model
=
Message)
async
def
message_handler
(
ctx
:
Context
,
sender
:
str
,
msg
:

Message):
ctx
.
logger
.
info
(
f
"Received message from
{
sender
}
:
{
msg.message
}
"
)
if
name
==
"**main**"
:
sigmar
.
run
()
Here, we have used the.on_message()
decorator to register themessage_handler()
coroutine function as a handler for incoming messages of typeMessage
.
Themessage_handler()
function takes three arguments:ctx
,sender
, andmsg
. Inside this function, we call thectx.logger.info()
method to log information about the received message, including the sender and message content.
We can now save the script.

The overall script for sigmar agent should be looking as follows:

remote_agents_sigmar.py from uagents import Agent , Context , Model from uagents . setup import fund_agent_if_low

class

Message (Model): message :

str

RECIPIENT_ADDRESS

"agent1q2kxet3vh0scsf0sm7y2erzz33cve6tv5uk63x64upw5g68kr0chkv7hw50"

sigmar

Agent (name = "sigmar" , port = 8000 , seed = "sigmar secret phrase" , endpoint = ["http://127.0.0.1:8000/submit"],)

fund_agent_if_low (sigmar.wallet. address ())

@sigmar . on_interval (period = 2.0) async

def

send_message (ctx : Context): await ctx . send (RECIPIENT_ADDRESS, Message (message = "hello there slaanesh"))

@sigmar . on_message (model = Message) async

def

message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ")

if

name

==

"**main**" : sigmar . run () We can now proceed by writing the script for agentslaanesh .

Slaanesh

1. Inremote_agents_slaanesh.py
2. script, import the necessary classes from theuagents
3. anduagents.setup
4. . Then, define the message structure for messages to be exchanged between the agents using theModel
5. class, as well as our second uAgent,slaanesh
6. , by providingname
7. ,seed
8. ,port
9. , andendpoint
10. . Make sure it has enough funds to register in the Almanac contract:
11. from
12. uagents
13. import
14. Agent
15. ,
16. Context
17. ,
18. Model
19. from
20. uagents
21. .
22. setup
23. import
24. fund_agent_if_low
25. class
26. Message
27. (
28. Model
29.):
30. message
31. :
32. str
33. slaanesh
34. =
35. Agent
36. (
37. name
38. =
39. "slaanesh"
40. ,
41. port
42. =
43. 8001
44. ,
45. seed
46. =
47. "slaanesh secret phrase"
48. ,
49. endpoint
50. =
51. [

"http://127.0.0.1:8001/submit"
],
)
fund_agent_if_low
(slaanesh.wallet.
address
())
Let's now define a function forslaanesh
to handle incoming messages and answering back to the sender:
@slaanesh
.
on_message
(model
=
Message)
async
def
message_handler
(
ctx
:
Context
,
sender
:
str
,
msg
:
Message):
ctx
.
logger
.
info
(
f
"Received message from
{
sender
}
:
{
msg.message
}
"
)
await
ctx
.
send
(sender,
Message
(message
=
"hello there sigmar"
))
if
name
==
"**main**"
:
slaanesh
.
run
()
Here, we have defined an asynchronousmessage_handler()
function for slaanesh to handle incoming messages from other uAgents. The function is decorated with.on_message()

120. , and it is triggered whenever a message of typeMessage
121. is received byslaanesh
122. . When a message is received, the handler function logs the sender's address and the content of the message. It then sends a response back to the sender using thectx.send()
123. with a new message. The response message contains theMessage
124. data model with a"hello there sigmar"
125. message.
126. Save the script.

The overall script forslaanesh should be looking as follows:

remote_agents_slaanesh.py from uagents . setup import fund_agent_if_low from uagents import Agent , Context , Model

class

Message (Model): message :

str

slaanesh

Agent (name = "slaanesh" , port = 8001 , seed = "slaanesh secret phrase" , endpoint = ["http://127.0.0.1:8001/submit"],)

fund_agent_if_low (slaanesh.wallet. address ())

@slaanesh . on_message (model = Message) async

def

message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ")

await ctx . send (sender, Message (message = "hello there sigmar"))

if

name

==

"**main**" : slaanesh . run ()

Run the scripts

In different terminal windows, first runslaanesh and thensigmar . They will register automatically in the Almanac contract using their funds. The received messages will print out in each terminal:

Terminal 1:python remote_agents_slaanesh.py

Terminal 2:python remote_agents_sigmar.py

The output will depend on the terminal:

- Sigmar
- :
- [sigmar]: Received message from agent1q2kxet3vh0scsf0sm7y2erzz33cve6tv5uk63x64upw5g68kr0chkv7hw50: hello there sigmar
- [sigmar]: Received message from agent1q2kxet3vh0scsf0sm7y2erzz33cve6tv5uk63x64upw5g68kr0chkv7hw50: hello there sigmar
- [sigmar]: Received message from agent1q2kxet3vh0scsf0sm7y2erzz33cve6tv5uk63x64upw5g68kr0chkv7hw50: hello there sigmar
- Slaanesh
- :
- [slaanesh]: Received message from agent1qdp9j2ev86k3h5acaayjm8tpx36zv4mjxn05pa2kwesspstzj697xy5vk2a: hello there slaanesh

- [slaanesh]: Received message from agent1qdp9j2ev86k3h5acaayjm8tpx36zv4mjxn05pa2kwesspstzj697xy5vk2a: hello there slaanesh
- [slaanesh]: Received message from agent1qdp9j2ev86k3h5acaayjm8tpx36zv4mjxn05pa2kwesspstzj697xy5vk2a: hello there slaanesh

Before we go on...

As we touched on before inRegister in Almanac ↗ , when the agent uses.run() function this tells theuagents library to register the agent to the Almanac. It's simple, agents initialize themselves, and register to a service which acts as a search engine for agents (the Almanac) then, when agents receive messages they can respond.

Conclusion

In this guide, we explored two different methods of communication for Agents using theuagents library:

- Local communication
- .
- Remote communication
- via the Almanac Contract.

Forlocal communication , we learned how to use theuagents library to create two agents,sigmar andslaanesh , and enable them to exchange messages with one another. We defined the message structure using theModel class and implemented message handlers for both agents. By running the script we observed their real-time message exchange.

Next, we delved intoremote communication , which facilitates interaction between agents through the Almanac Contract. This method requires registering the agents in the Almanac Contract and querying forHTTP endpoints for communication. By running the scripts separately, we could observe the real-time messages exchange, fostering a decentralized network of interacting agents.

With this, we suspect you're ready to start building agents, as part of multi agent system the Almanac allows; awesome. If you want to go further though, take a look at themessage verification↗ andsending tokens ↗ , after-all you do want to be sure you are speaking to who you think you are, and agents getting paid is awesome.

Was this page helpful?

Agents address Agent Handlers (on_...)