

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 the [Almanac Contract](#) ↗
- .

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

i Local communication is important for debugging purposes.

AI 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 import `Agent`
3. `, Context`
4. `, Bureau`
5. `, and Model`
6. from the `uagents` library and we then define the message structure for messages to be exchanged between the agents using the class `Model`
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. and `slaanesh`
30. `, with name`
31. and `seed`
32. parameters:
33. `sigmar`
34. `=`
35. `Agent`
36. `(name`
37. `=`
38. `"sigmar"`

```

39. , seed
40. =
41. "sigmar recovery phrase"
42. )
43. slaanesh
44. =
45. Agent
46. (name
47. =
48. "slaanesh"
49. , seed
50. =
51. "slaanesh recovery phrase"
52. )
53. In this example we're running multiple agents from one file.
54. Let's now definesigmar
55. 's behaviors. We need to define a function forsigmar
56. to send messages toslaanesh
57. periodically:
58. @sigmar
59. .
60. on_interval
61. (period
62. =
63. 3.0
64. )
65. async
66. def
67. send_message
68. (
69. ctx
70. :
71. Context):
72. await
73. ctx
74. .
75. send
76. (slaanesh.address,
77. Message
78. (message
79. =
80. "hello there slaanesh"
81. ))
82. We can use the.on_interval()
83. decorator to define a coroutinesend_message()
84. function that will be called every 3 seconds. The coroutine function sends a message toslaanesh
85. using thectx.send()
86. method of theContext
87. object.
88. We then need to define asigmar_message_handler()
89. function forsigmar
90. to manage incoming messages:
91. @sigmar
92. .
93. on_message
94. (model
95. =
96. Message)
97. async
98. def
99. sigmar_message_handler
100. (
101. ctx
102. :
103. Context
104. ,
105. sender
106. :

```

```

107. str
108. ,
109. msg
110. :
111. Message):
112. ctx
113. .
114. logger
115. .
116. info
117. (
118. f
119. "Received message from
120. {
121. sender
122. }
123. :
124. {
125. msg.message
126. }
127. "
128. )
129. This defines the coroutine function sigmar_message_handler()
130. that serves as a message handler for sigmar
131. . It is triggered whenever sigmar
132. receives a message of type Message
133. .
134. Let's now define the behavior of our second agent, slaanesh
135. :
136. @slaanesh
137. .
138. on_message
139. (model
140. =
141. Message)
142. async
143. def
144. slaanesh_message_handler
145. (
146. ctx
147. :
148. Context
149. ,
150. sender
151. :
152. str
153. ,
154. msg
155. :
156. Message):
157. ctx
158. .
159. logger
160. .
161. info
162. (
163. f
164. "Received message from
165. {
166. sender
167. }
168. :
169. {
170. msg.message
171. }
172. "
173. )
174. await

```

```

175. ctx
176. .
177. send
178. (sigmar.address,
179. Message
180. (message
181. =
182. "hello there sigmar"
183. ))
184. Same assignmar
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 agent1q0mau8vkmg78xx0sh8cyl4tpl4ktx94ppq2e94cylu6haugt2hd7j9vequ7: hello there
sigmar [ slaanesh]: Received message from agent1qww3ju3h6kfcuqf54gkghvt2pqe8qp97a7nzm2vp8plfxflc0epzcjsv79t:
hello there slaanesh [sigmar]: Received message from
agent1q0mau8vkmg78xx0sh8cyl4tpl4ktx94ppq2e94cylu6haugt2hd7j9vequ7: hello there sigmar [ slaanesh]: Received
message from agent1qww3ju3h6kfcuqf54gkghvt2pqe8qp97a7nzm2vp8plfxflc0epzcjsv79t: hello there slaanesh [sigmar]:
Received message from agent1q0mau8vkmg78xx0sh8cyl4tpl4ktx94ppq2e94cylu6haugt2hd7j9vequ7: hello there sigmar

```

AI 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 for sigmar .

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
62. =
```

```

63. [
64. "http://127.0.0.1:8000/submit"
65. ],
66. )
67. fund_agent_if_low
68. (sigmar.wallet.
69. address
70. ())
71. On the Fetch.ai testnet, you can use the fund_agent_if_low
72. 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 of sigmar
73. 's wallet is low and funds it if necessary.
74. We are ready to define sigmar
75. 's behaviors. Let's start with a function for sigmar
76. to send messages:
77. @sigmar
78. .
79. on_interval
80. (period
81. =
82. 2.0
83. )
84. async
85. def
86. send_message
87. (
88. ctx
89. :
90. Context):
91. await
92. ctx
93. .
94. send
95. (RECIPIENT_ADDRESS,
96. Message
97. (message
98. =
99. "hello there slaanesh"
100. )
101. Here, the on_interval()
102. decorator schedules the send_message()
103. function to be run every 2 seconds. Inside the function, there is an asynchronous call indicated by the ctx.send()
104. method. This call sends a message with the content "hello there slaanesh"
105. to the RECIPIENT_ADDRESS
106. .
107. We then need to define a function for sigmar
108. to handle incoming messages from other agents:
109. @sigmar
110. .
111. on_message
112. (model
113. =
114. Message)
115. async
116. def
117. message_handler
118. (
119. ctx
120. :
121. Context
122. ,
123. sender
124. :
125. str
126. ,
127. msg
128. :
129. Message):

```

```

130. ctx
131. .
132. logger
133. .
134. info
135. (
136. f
137. "Received message from
138. {
139. sender
140. }
141. :
142. {
143. msg.message
144. }
145. "
146. )
147. if
148. name
149. ==
150. "main"
151. :
152. sigmar
153. .
154. run
155. ()
156. Here, we have used the.on_message()
157. decorator to register themessage_handler()
158. coroutine function as a handler for incoming messages of typeMessage
159. .
160. Themessage_handler()
161. function takes three arguments:ctx
162. ,sender
163. , andmsg
164. . Inside this function, we call thectx.logger.info()
165. method to log information about the received message, including the sender and message content.
166. 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. [

```

```

52. "http://127.0.0.1:8001/submit"
53. ],
54. )
55. fund_agent_if_low
56. (slaanesh.wallet.
57. address
58. ())
59. Let's now define a function for slaanesh
60. to handle incoming messages and answering back to the sender:
61. @slaanesh
62. .
63. on_message
64. (model
65. =
66. Message)
67. async
68. def
69. message_handler
70. (
71. ctx
72. :
73. Context
74. ,
75. sender
76. :
77. str
78. ,
79. msg
80. :
81. Message):
82. ctx
83. .
84. logger
85. .
86. info
87. (
88. f
89. "Received message from
90. {
91. sender
92. }
93. :
94. {
95. msg.message
96. }
97. "
98. )
99. await
100. ctx
101. .
102. send
103. (sender,
104. Message
105. (message
106. =
107. "hello there sigmar"
108. ))
109. if
110. name
111. ==
112. "main"
113. :
114. slaanesh
115. .
116. run
117. ()
118. Here, we have defined an asynchronous message_handler()
119. 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 in [Register in Almanac ↗](#), when the agent uses `run()` function this tells the `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 AI Agents using the `theuagents` library:

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

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

Next, we delved into remote communication, which facilitates interaction between agents through the Almanac Contract. This method requires registering the agents in the Almanac Contract and querying for HTTP 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 the [message verification ↗](#) and [sending 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?

[Registering in the Almanac Contract](#) [How to book a table at a restaurant using agents](#)