Agent Chat Protocol

This guide walks you through the implementation of agent-to-agent communication using protocols. It includes registering protocols in an Almanac contract and understanding their metadata, which is stored as manifests.

What is Protocol?

A protocol is a self-contained collection of message handlers. Its digest ensures that the protocol reliably supports the intended messages, serving as a unique identifier for the specific implementation.

When to Use Protocols

Protocols are designed for modular and reusable communication behavior. They are essential for scaling ecosystems by enabling interoperability.

- Reusable Communication
- : Support predefined message structures to connect with existing systems effortlessly.
- Compatibility
- : Following protocols ensures seamless communication between agents.
- Scalability
- : Standardized interactions allow the ecosystem to grow and integrate new agents easily.

What Are Chat Protocols?

Agent chat protocols define the rules and structure for communication between agents. They ensure that messages exchanged between agents are well-structured, predictable, and interoperable. In the context of agent-based systems, chat protocols specify:

- The types of messages agents can send and receive.
- The sequence of interactions between agents.
- How messages are processed and responded to.

In a multi-agent system, agents are identified by their interactive behavior—what output they provide in response to given input. Code unrelated to interactions is irrelevant, as it does not affect observable behavior. Therefore, only protocols have manifests, not agents.

Overview

In this implementation, two agents,InitiatorAgent and aResponderAgent , communicate using predefined protocols. The agents exchange messages, and their protocols and models are registered in an Almanac contract. The metadata is stored as a manifest describing the protocol, models, and interactions.

Protocols and Communication

Message Models

- RequestMessage
- : Represents the request initiated by the sender.
- ResponseMessage
- : Represents the response sent back by the receiver.
- Both models ensure structured and predictable data exchange.

class

RequestMessage (Model): text :

str

class

ResponseMessage (Model): text :

str

Initialize Agents

Two agents,InitiatorAgent and ResponderAgent, are created with unique seeds for identity.

initiator_agent

Agent (name = "InitiatorAgent" , seed = "initiator recovery phrase" ,)

responder_agent

Agent (name = "ResponderAgent" , seed = "responder recovery phrase" ,)

Define Chat Protocols

Protocols specify how agents interact. TheSimpleProtocol_Initiator is for the initiator, andSimpleProtocol_Responder is for the responder.

initiator_protocol

Protocol (name = "SimpleProtocol_Initiator" , version = "0.1.0") responder_protocol =

Protocol (name = "SimpleProtocol_Responder" , version = "0.1.0")

Implement Protocol Handlers

Handlers define the interaction logic:

1. Initiator Sends a Message Periodically:

@initiator_protocol . on_interval (period = 3.0) async

def

initiator_send_message (ctx : Context): await ctx . send (responder_agent.address, RequestMessage (text = "Hello there from Initiator!")) 1. Responder Handles the Request and Replies:

@responder_protocol . on_message (RequestMessage, replies = ResponseMessage) async

def

responder_handle_message (ctx : Context ,

sender :

str ,

msg : RequestMessage): ctx . logger . info (f "Received message from { sender } : { msg.text } ") await ctx . send (sender, ResponseMessage (text = "Hello there from Responder!")) 1. Initiator Handles the Response:

@initiator_protocol . on_message (model = ResponseMessage) async

def

initiator_handle_response (ctx : Context ,

sender :

str ,

msg : ResponseMessage): ctx . logger . info (f "Received response from { sender } : { msg.text } ")

Bureau Initialization

TheBureau facilitates the agents' execution and connects them via an endpoint.

bureau

Bureau (endpoint = ["http://127.0.0.1:8000/submit"]) bureau . add (initiator_agent) bureau . add (responder_agent)

Main Function

Agents include their respective protocols and register their metadata manifest with the Almanac contract.

if

name

==

'**main**' : initiator_agent . include (initiator_protocol, publish_manifest = True) responder_agent . include (responder_protocol, publish_manifest = True) bureau . run ()

Metadata Manifest

This is a registered manifest for agents in the Almanac system, defining communication protocols between the initiator and responder. It specifies message models for both requests and responses. When protocols are registered, metadata is stored in the Almanac as a manifest , Published Protocol Manifest looks something like this

- forinitiator_agent
- published manifest looks like:

{ "version" :

"1.0" , "metadata" : { "name" :

"SimpleProtocol_Initiator" , "version" :

"0.1.0" , "digest" :

"proto:2a34b5504c58f43b2932cdd73358cebe0b668ea10e6796abba3dec8a4c50f25b" } , "models" : [{ "digest" :

"model:465d2d900b616bb4082d4d7fcd9cc558643bb1b9b45660a7f546d5b5b5c0aba5" , "schema" : { "properties" : { "text" : { "title" :

"Text" , "type" :

"string" } } , "required" : ["text"] , "title" :

"ResponseMessage" , "type" :

"object" } }] , "interactions" : [] } * forresponder_agent * published manifest looks like:

{ "version" :

"1.0" , "metadata" : { "name" :

"SimpleProtocol_Responder" , "version" :

"0.1.0" , "digest" :

"proto:c93ed21a1091272c178c4f6b05619405204e6458294b4a6ee080299bf20e619a" } , "models" : [{ "digest" :

"model:ae2de187153cc7a80641a52927aa2852a820cd56bbbdb8671a0d1e643472f9b7" , "schema" : { "properties" : { "text" : { "title" :

"Text" , "type" :

"string" } } , "required" : ["text"] , "title" :

"RequestMessage" , "type" :

"object" } } , { "digest" :

"model:465d2d900b616bb4082d4d7fcd9cc558643bb1b9b45660a7f546d5b5b5c0aba5" , "schema" : { "properties" : { "text" : { "title" :

"Text" , "type" :

"string" } } , "required" : ["text"] , "title" :

"ResponseMessage" , "type" :

"object" } }] , "interactions" : [{ "type" :

"normal" , "request" :

"model:ae2de187153cc7a80641a52927aa2852a820cd56bbbdb8671a0d1e643472f9b7" , "responses" : ["model:465d2d900b616bb4082d4d7fcd9cc558643bb1b9b45660a7f546d5b5b5c0aba5"] }] } Last updated on January 20, 2025

Was this page helpful?

You can also leave detailed feedback on Github

Multi-file agent pipeline for AI Engine: Hugging face API to create a multi agent pipeline Search Agent

On This Page