Aries Cloud Agent Python: Architectural Deep Dive

As deep as you want to go...

Remember the Hyperledger Code of Conduct

Anti-Trust Policy:

Linux Foundation meetings involve participation by industry competitors, and it is the intention of the Linux Foundation to conduct all of its activities in accordance with applicable antitrust and competition laws. It is therefore extremely important that attendees adhere to meeting agendas, and be aware of, and not participate in any activities that are prohibited under applicable US state, federal or foreign antitrust and competition laws.

Examples of types of actions that are prohibited at Linux Foundation meetings and in connection with Linux Foundation activities are described in the Linux Foundation Antitrust Policy available at http://www.linuxfoundation.org/antitrust-policy. If you have questions about these matters, please contact your company counsel, or if you are a member of the Linux Foundation, feel free to contact Andrew Updegrove of the firm of Gesmer Updegrove LLP, which provides legal counsel to the Linux Foundation.

Goal: Architectural Introduction

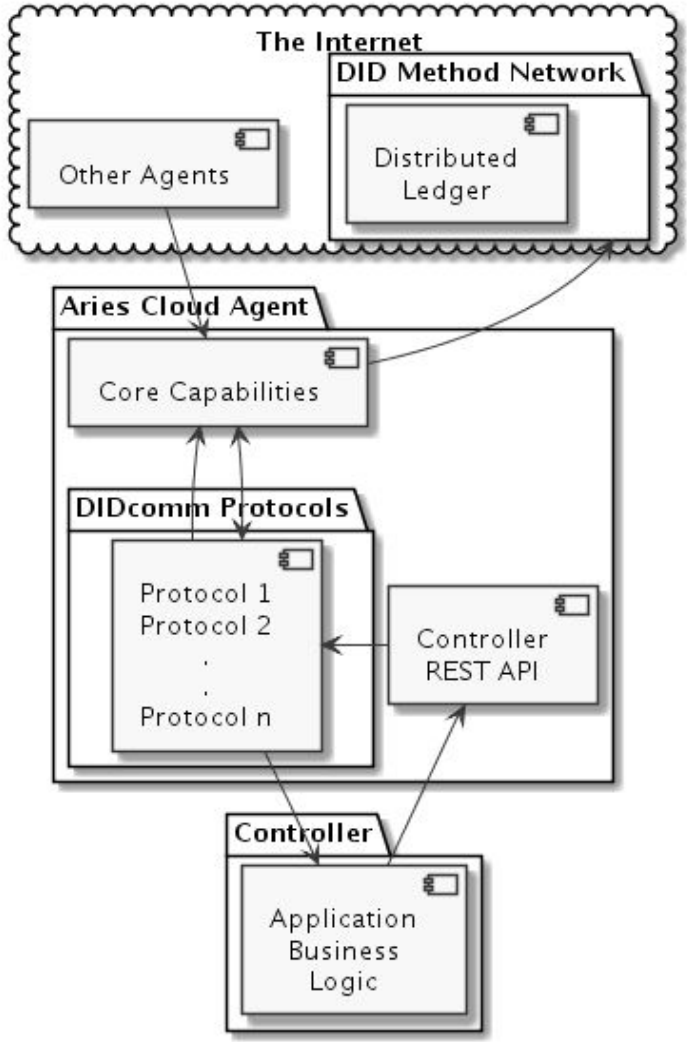
Describe the architecture for Aries Cloud Agent ***controller*** developers first (and quickly), and then delve deeper for developers interested in digging into the ***agent*** code.

Agenda

- Overall Architecture
- Documentation and Demos
- Deployment and Command Line Parameters
- Controller: Events and Administrative API
- Message Flow: Inbound and Outbound
- Protocol Implementations - Writing Your Own
- Pluggable Agent Components
- Developer Concerns
- Other Topics

Overall Architecture

- The agent - built from PyPi and deployed “as is”
  - Configured via command line parameters
  - Interacts with other agents via pluggable transports
  - Manages storage, ledger with pluggable implementations
  - Manages messages and protocol state
  - Invokes protocols (configurable set)
  - Driven by a controller
    - Sends events to controller
    - Exposes an HTTP JSON administrative API to controller
- The controller - business logic for instance of agent instance
  - Receives events from agent
  - Sends requests to agent using HTTP JSON administrative API



Documentation and Demos

- README
- Developer ReadMe
- Getting Started Guide
- Demos
  - IIWBook
  - Alice/Faber Command Line
  - Alice/Faber OpenAPI
  - Performance run, Performance with Routing Agent Demo
  - Adding ACME Exercise
- ReadTheDocs - coming soon
  - Simple generation is not useful - laundry list of functions
  - Need to balance effort to manage the list to be useful vs. automating generation of the documentation

Deployment and Command Line Parameters

- `aca-py` executable uses many command line arguments - documented here
- Intention is a parameter set that can be tuned for each agent deployment use case
  - Example: Demos use a set that define a specific behaviour for invitations, credential exchange
  - Wrapper scripts allow you to pass in parameters at deployment time
  - Docker invocations provide explicit settings as the basis for production deploys
  - Openshift examples cover cloud native deployments - e.g. Kubernetes
- A number of categories of parameters...

Parameter Categories

- Transports: inbound, outbound, endpoint
- Logging/debugging settings
- Label: self-attested agent name
- Wallet implementation and related info
- Ledger parameters (e.g. genesis URL, etc.)
- For controller: Admin API configuration
  - URL
  - Security selection
- Protocol automation flags
  - E.g. accept-invites, accept-requests
  - Seed to create on startup
  - Create invitation on startup
- Add timing information to messaging
- Optional protocols to load
- From controller: Event webhook URL

- Future direction: git- or docker-style approach - subcommands and parameters
  - run - running an agent
  - wallet - creating/configuring a wallet - less tied to running an agent
  - ledger - setup ledger (e.g. set genesis txns)

Controller: Events and Administrative API

- Listen for HTTP requests on an webhook URL
  - Process event
  - Optionally pass to external system (person, legacy system, etc.) to get “next step” decision
  - Optionally respond with an HTTP request to the Agent’s Administrative API
- Take inputs from other sources
  - Process input
  - Optionally initiate agent protocols with an HTTP request to the Agent’s Administrative API
- Simple example: Alice/Faber command line API
  - ***Simple***: Both agents know about each other
  - Deploys agent as a sub-process, initializes and waits at command line for user input
  - Process user input to initiate protocol invocations
  - Listen for, receive, process and respond to protocol events
    - Report event received and automatically process (based on command line parameters)

Your Production Controller Repo

- Contains code for controller
  - Generally created from scratch or copied (**not** forked) from an existing implementation
- Contains deployment configuration for your instance of an aries-cloudagent-python
  - Do ***not*** fork the aries-cloudagent-python as the basis of your controller
  - Pull in from PyPi and deploy
    - Recommendation: Subscribe to the repo for release notifications
- Examples (with docker and openshift deployment configs)
  - Django: bcgov/indy-email-verification
  - Django: bcgov/iiwbook

Controller Events

- Internal state is maintained for all protocols
- Every time a state transition occurs in a protocol:
  - Internal representation of that protocol's state is serialized
  - Sent to the controller as a webhook
- Currently all state transitions are sent to the controller.
  - In a future enhancement, granular webhook registration will be allowed.
- The controller can use the data in the state transitions to persist relevant information and respond to event by making requests to the agent's administrative API.

Controller: Administrative API

- During your development - available as a generated OpenAPI (Swagger) web interface
  - OpenAPI demo available here
- The API must either be:
  - Explicitly run in insecure mode for development using `--admin-insecure-mode` or
  - Given an API key using `--admin-api-key`
    - Must be present in the X-API-Key header in all admin api requests
- Each protocol adds Admin API endpoints
- Externally loaded protocols can also add their own admin endpoints
  - Dynamic OpenAPI
  - Externally loaded protocols are Python modules

Message Flow: Inbound

1. Message hits the registered inbound transport
2. Passed to message router in the `conductor`
    a. Deserialized into data object
    b. Associate message with internal connection state
3. Passed into `dispatcher`
    a. Hydrate an internal `message` object
    b. Process id/thread decorator
        i. Send problem reports
    c. Process other decorators (skip as appropriate)
    d. Call handler - protocol within messaging folder
    e. Handle sending of outbound messages for handlers

Message Flow: Outbound

1. Message handlers and admin route handlers obtain a Responder object with methods to dispatch an outbound message, usually to a registered connection ID
2. The Dispatcher or AdminServer returns the outbound message to the Conductor
    a. may first adjust message decorators, for example by adding the ~timing decorator
3. The Conductor packs the message according to the routing and recipient keys in the connection's DIDDoc, and populates the endpoint URL
4. When return routing is active the Conductor may return the message directly to an open socket held by an inbound transport
5. Otherwise the packed message and endpoint are passed to the OutboundTransportHandler
    a. determines the appropriate transport based on the URL scheme
    b. connects to the endpoint and delivers the message

Protocol Implementations - Writing Your Own

- Reason:
  - Contribute: Implement Aries RFCs - standard/core protocols
  - Implement your own use case-specific protocol
    - Example: Indy-Catalyst "Register Issuer" protocol
- Process:
  - Define as module including version, requirements
  - Define, implement module
    - Register API endpoints
    - Define messages, handlers and models
    - Define tests
- Protocol versioning
  - General approach: Have multiple copies of protocol, deprecated and new version

Pluggable Agent Components

- Reason
  - You want to be able to add support for other implementations for your use cases
- Pluggable implementations for:
  - Inbound, Outbound Transports: currently HTTP and WebSockets
  - Wallets - Indy entities (DIDs, schema, cred defs, etc.)
    - Basic - in memory
    - Indy-SDK - SQLite, Postgres
  - Storage - non-secrets (anything else)
    - Basic - in memory
    - Indy-SDK - SQLite, Postgre
  - Future(?): Ledger Integration

Other components

- aries_cloudagent - all injected in a generic way (but less likely to have alternate implementations)
  - admin - admin HTTP JSON interface
  - cache - basic in-memory cache of (mostly for now) connection records
  - config - command line parser, runtime configuration classes
  - holder / issuer / verifier - (currently) indy implementations of ledger interactions for types of participants

Developer Concerns

- Choosing the right controller stack
  - Language? Python, Node/JS, Java, etc.
  - Web development framework? Django, vue, etc.
  - Legacy integration approach?
- `--debug` supporting IDE debugging integration while running in Docker
  - Demo
  - Currently supports Visual Studio debugging (including VSCode) with ptvsd
    - Hint Hint: Could use guidance on supporting other approaches

Other Topics

- Routing
  - See performance demo with option to run with routing node
  - Provisioning routing agent
  - Configuring routing agent per connection