Title (Ex. 0000: Deliverable Name)

- Authors: your name – email is optional
- Deliverable Type: *Spec - Protocol*
- Status: PROPOSED
- Since: YYYY-MM-DD (date you submit your PR)
- Status Note: (explanation of current status)

- Supersedes: (link to anything this ToIP Deliverable supersedes)
- Start Date: YYYY-MM-DD (date you started working on this idea)
- Tags: (see ../../../tags.md) ## Summary

One paragraph explanation of the feature.

> If the RFC you are proposing is **NOT** a protocol, please use this template as a starting point.
>
> When completing this template and before submitting as a PR, please remove the template text in sections (other than **Implementations**). The implementations section should remain as is.

Motivation

Why are we doing this? What use cases does it support? What is the expected outcome?

Tutorial

Name and Version

Name and Version

Specify the official name of the protocol and its version, e.g., "My Protocol 0.9".

Protocol names are often either lower_snake_case or kebob-case. The non-version components of the protocol named are matched exactly.

URI: https://didcomm.org/lets_do_lunch//

Message types and protocols are identified with special URIs that match certain conventions. See Message Type and Protocol Identifier URIs for more details.

The version of a protocol is declared carefully. See Semver Rules for Protocols for details.

Key Concepts

This is short–a paragraph or two. It defines terms and describes the flow of the interaction at a very high level. Key preconditions should be noted (e.g., "You can't issue a credential until you have completed the *connection* protocol first"), as well as ways the protocol can start and end, and what can go wrong.

The section might also talk about timing constraints and other assumptions. After reading this section, a developer should know what problem your protocol solves, and should have a rough idea of how the protocol works in its simpler variants.

Roles

> See this note for definitions of the terms "role", "participant", and "party".

Provides a formal name to each role in the protocol, says who and how many can play each role, and describes constraints associated with those roles (e.g., "You can only issue a credential if you have a DID on the public ledger"). The issue of qualification for roles can also be explored (e.g., "The holder of the credential must be known to the issuer").

The formal names for each role are important because they are used when agents discover one another's capabilities; an agent doesn't just claim that it supports a protocol; it makes a claim about which *roles* in the protocol it supports. An agent that supports credential issuance and an agent that supports credential holding may have very different features, but they both use the *credential-issuance* protocol. By convention, role names use lower-kebab-case and are compared case-sensitively.

States

This section lists the possible states that exist for each role. It also enumerates the events (often but not always messages) that can occur, including errors, and what should happen to state as a result. A formal representation of this information is provided in a *state machine matrix*. It lists events as columns, and states as rows; a cell answers the question, "If I am in state X (=row), and event Y (=column) occurs, what happens to my state?" The Tic Tac Toe example is typical.

Choreography Diagrams from BPMN are good artifacts here, as are PUML sequence diagrams and UML-style state machine diagrams. The matrix form is nice because it forces an exhaustive analysis of every possible event. The diagram styles are often simpler to create and consume, and the PUML and BPMN forms have the virtue that they can support line-by-line diffs when checked in with source code. However, they don't offer an easy way to see if all possible flows have been considered; what they may NOT describe isn't obvious. This– and the freedom from fancy tools–is why the matrix form is used in many early RFCs. We leave it up to the community to settle on whether it wants to strongly recommend specific diagram types.

The formal names for each state are important, as they are used in `ack`s and `problem-report`s). For example, a `problem-report` message declares which state the sender arrived at because of the problem. This helps other participants

to react to errors with confidence. Formal state names are also used in the agent test suite, in log messages, and so forth.

By convention, state names use lower-kebab-case. They are compared case-sensitively.

State management in protocols is a deep topic. For more information, please see State Details and State Machines.

Messages

This section describes each message in the protocol. It should also note the names and versions of messages from other message families that are adopted by the protocol (e.g., an `ack` or a `problem-report`). Typically this section is written as a narrative, showing each message type in the context of an end-to-end sample interaction. All possible fields may not appear; an exhaustive catalog is saved for the "Reference" section.

Sample messages that are presented in the narrative should also be checked in next to the markdown of the RFC, in DIDComm Plaintext format.

The *message* element of a message type URI are typically lower_camel_case or lower-kebab-case, matching the style of the protocol. JSON items in messages are lower_camel_case and inconsistency in the application of a style within a message is frowned upon by the community.

Adopted Messages

Many protocols should use general-purpose messages such as `ack` and `problem-report`) at certain points in an interaction. This reuse is strongly encouraged because it helps us avoid defining redundant message types–and the code to handle them–over and over again (see DRY principle).

However, using messages with generic values of `@type` (e.g., `"@type": "https://didcomm.org/notification/1.0/ack"`) introduces a challenge for agents as they route messages to their internal routines for handling. We expect internal handlers to be organized around protocols, since a protocol is a discrete unit of business value as well as a unit of testing in our agent test suite. Early work on agents has gravitated towards pluggable, routable protocols as a unit of code encapsulation and dependency as well. Thus the natural routing question inside an agent, when it sees a message, is "Which protocol handler should I route this message to, based on its @type?" A generic `ack` can't be routed this way.

Therefore, we allow a protocol to **adopt** messages into its namespace. This works very much like python's `from module import symbol` syntax. It changes the `@type` attribute of the adopted message. Suppose a `rendezvous` protocol is identified by the URI `https://didcomm.org/rendezvous/2.0`, and its definition announces that it has adopted generic 1.x `ack` messages. When such

`ack` messages are sent, the `@type` should now use the alias defined inside the namespace of the `rendezvous` protocol:

diff on @type caused by adoption

Adoption should be declared in an "Adopted" subsection of "Messages". When adoption is specified, it should include a **minimum adopted version** of the adopted message type: "This protocol adopts `ack` with version >= 1.4". All versions of the adopted message that share the same major number should be compatible, given the semver rules that apply to protocols.

Constraints

Many protocols have constraints that help parties build trust. For example, in buying a house, the protocol includes such things as commission paid to realtors to guarantee their incentives, title insurance, earnest money, and a phase of the process where a home inspection takes place. If you are documenting a protocol that has attributes like these, explain them here. If not, the section can be omitted.

Reference

All of the sections of reference are optional. If none are needed, the "Reference" section can be deleted.

Messages Details

Unless the "Messages" section under "Tutorial" covered everything that needs to be known about all message fields, this is where the data type, validation rules, and semantics of each field in each message type are details. Enumerating possible values, or providing ABNF or regexes is encouraged. Following conventions such as those for date- and time-related fields can save a lot of time here.

Each message type should be associated with one or more roles in the protocol. That is, it should be clear which roles can send and receive which message types.

If the "Tutorial" section covers everything about the messages, this section should be deleted.

Examples

This section is optional. It can be used to show alternate flows through the protocol.

Collateral

This section is optional. It could be used to reference files, code, relevant standards, oracles, test suites, or other artifacts that would be useful to an imple-

menter. In general, collateral should be checked in with the RFC.

Localization

If communication in the protocol involves humans, then localization of message content may be relevant. Default settings for localization of all messages in the protocol can be specified in an `l10n.json` file described here and checked in with the RFC. See "Decorators at Message Type Scope" in the Localization RFC.

Codes Catalog

If the protocol has a formally defined catalog of codes (e.g., for errors or for statuses), define them in this section. See "Message Codes and Catalogs" in the Localization RFC.

Drawbacks

Why should we *not* do this?

Rationale and alternatives

- Why is this design the best in the space of possible designs?
- What other designs have been considered and what is the rationale for not choosing them?
- What is the impact of not doing this?

Prior art

Discuss prior art, both the good and the bad, in relation to this proposal. A few examples of what this can include are:

- Does this feature exist in other SSI ecosystems and what experience have their community had?
- For other teams: What lessons can we learn from other attempts?
- Papers: Are there any published papers or great posts that discuss this? If you have some relevant papers to refer to, this can serve as a more detailed theoretical background.

This section is intended to encourage you as an author to think about the lessons from other implementers, provide readers of your proposal with a fuller picture. If there is no prior art, that is fine - your ideas are interesting to us whether they are brand new or if they are an adaptation from other communities.

Note that while precedent set by other communities is some motivation, it does not on its own motivate an enhancement proposal here. Please also take into consideration that Aries sometimes intentionally diverges from common identity features.

Unresolved questions

- What parts of the design do you expect to resolve through the enhancement proposal process before this gets merged?
- What parts of the design do you expect to resolve through the implementation of this feature before stabilization?
- What related issues do you consider out of scope for this proposal that could be addressed in the future independently of the solution that comes out of this doc?

Implementations

> NOTE: This section should remain in the RFC as is on first release. Remove this note and leave the rest of the text as is. Template text in all other sections should be removed before submitting your Pull Request.

The following lists the implementations (if any) of this RFC. Please do a pull request to add your implementation. If the implementation is open source, include a link to the repo or to the implementation within the repo. Please be consistent in the "Name" field so that a mechanical processing of the RFCs can generate a list of all RFCs supported by an Aries implementation.

Implementation Notes may need to include a link to test results.

Name / Link	Implementation Notes