Meshery Adapter Extension Point

Making the Extension Point Generic

Status: **Draft** | *Under Review* | *Approved*

Status: **Draft** | *Under Review* | *Approved*

Design Prologue

Meshery, as a comprehensive cloud-native management plane, orchestrates and manages various Cloud-native resources across diverse environments. The current adapter extension point, which uses a `.proto` file (`meshes.proto`), is limited in its ability to scale and adapt to new requirements. This proposal outlines a plan to refactor the adapter extension point to be more generic and reusable, addressing the current system's rigidity and improving its flexibility and scalability.

Design Goals and Objectives

The existing `meshes.proto` file defines a fixed set of operations that adapters can perform, which results in several key issues:

1. **Limited Scalability:**
   Adapters must conform to a predefined set of operations, which restricts the addition of new functionalities and technologies. This rigidity hinders the ability to integrate new Cloud-native resources effectively.
2. **Inflexibility:**
   Adapters are constrained by the fixed operations defined in `meshes.proto`. Adapting to new or unique capabilities requires changes to the `.proto` file, which can be cumbersome and error-prone.

FAQ

- **Common question**
  - And its answer

Functional Architecture

To address these issues, the proposal recommends the following changes:

Rename `meshes.proto` to `adapters.proto`:

Renaming the file to `adapters.proto` aligns with the broader scope of managing diverse Cloud-native resources and avoids the specific focus on service meshes.

Use `google.protobuf.Any` for Parameters and Results:

Replace hardcoded operations with a more flexible approach using `google.protobuf.Any` for operation parameters and results. This allows each adapter to define its own message types, facilitating greater flexibility and scalability.

Revised `adapters.proto` Example:

```
syntax = "proto3";

import "google/protobuf/any.proto";

package meshery;

service MesheryAdapter {
  rpc RegisterAdapter(RegisterRequest) returns (RegisterResponse);
  rpc GetCapabilities(Empty) returns (CapabilitiesResponse);
  rpc ExecuteOperation(OperationRequest) returns (OperationResponse);
  rpc StreamEvents(EventsRequest) returns (stream EventsResponse) {}
}

message EventsRequest {}

message EventsResponse {
    EventType event_type = 1;
```

```
    string summary = 2;
    string details = 3;
    string operation_id = 4;
    string probable_cause = 5;
    string suggested_remediation = 6;
    string error_code = 7;
    string component = 8;
    string component_name = 9;
}


message RegisterRequest {
  string adapter_name = 1;
  string adapter_version = 2;
}

message RegisterResponse {
  bool success = 1;
  string message = 2;
}

message Empty {}

message CapabilitiesResponse {
  repeated string capabilities = 1;
}

message OperationRequest {
  string capability = 1;
  google.protobuf.Any params = 2;
}

message OperationResponse {
  bool success = 1;
  string message = 2;
  google.protobuf.Any result = 3;
```

```
}
```

Benefits:

- **Scalability:** Adapters can introduce and evolve their own capabilities independently of Meshery's core definitions.
- **Flexibility:** The `Any` type supports diverse operations, accommodating various types of Cloud-native resources and technologies.
- **Reduced Maintenance:** Minimizes the need for frequent `.proto` file updates and reduces the maintenance burden associated with ensuring backward compatibility.

Addition of Capability Struct:

The `Capability` construct should contain more information. It should at least contain description, accepted parameters, and their types. This will enable the UI to just get the list of capabilities, and render them in the UI. We could even embed an RJSF schema within the `capability` construct to dynamically render them in the UI. Every capability should have a list of params, and their accepted types.

For instance:

```
message Param {
string name = 1
string json_schema = 2
}

message Capability {
string id = 1
string description = 2
string display_name = 3
```

```
repeated Param required_params = 4
repeated Param optional_params = 5
optional string ui_rjsf_schema = 6
}
```

Dynamic Capability Discovery:

Implement dynamic capability discovery so that Meshery can automatically detect and present the capabilities of each adapter. Adapters should register their capabilities during the `RegisterAdapter` call, which will be reflected in the Meshery UI.
Example:

```
message CapabilitiesResponse {
  repeated string capabilities = 1;
}
```

Versioning and Compatibility:
Introduce versioning in the adapter registration process to handle different versions of adapters. This approach ensures compatibility and facilitates smooth transitions as adapters evolve.
Example:

```
message RegisterRequest {
  string adapter_name = 1;
  string adapter_version = 2;
}
```

2. **Interoperability and Extensibility:**
   The proposed changes enhance interoperability, allowing adapters to be implemented in any language that supports gRPC and Protocol Buffers. This broadens Meshery's integration capabilities and supports future enhancements, such as dynamic loading and unloading of adapters.

Implementation Plan

1. **Proto File Migration:**
   - Rename `meshes.proto` to `adapters.proto`.
   - Update all references to the old proto file in the Meshery codebase.
   - Implement the revised `adapters.proto` with `Any` type for operation parameters and results.
   - Move the proto file to meshery/schemas repository
2. **Meshery Server Update:**
   - Refactor the Meshery server to support the new `adapters.proto` definitions.
   - Implement dynamic capability discovery and versioning support.
3. **Adapter Refactoring:**
   - Update existing adapters to use the new `adapters.proto` file.
   - Ensure that adapters define their own Protobuf messages and maintain backward compatibility or provide migration paths.
   - Upgrade the list of
4. **UI Update:**
   - Modify the Meshery UI to reflect dynamic discovery of adapter capabilities and provide interfaces for interacting with generic operations.
5. **Testing and Validation:**
   - Conduct extensive testing to validate the new implementation and ensure it meets the scalability and flexibility requirements.
   - Verify that existing adapters function correctly with minimal changes and assess overall system performance.

Refactoring the Meshery Adapter extension point to be more generic and scalable is crucial for maintaining the project's flexibility and extensibility. By leveraging google.protobuf.Any

and introducing dynamic capability discovery, Meshery will be better equipped to manage a diverse array of Cloud-native resources. This proposal aims to rejuvenate the existing adapter ecosystem, ensure compatibility, and stimulate renewed interest and contributions.

Feedback and collaboration are encouraged to refine this proposal and ensure it aligns with the project's goals and community needs.

Architecture Diagram

<to-be-updated>

Sequence Diagram

<to-be-updated>

System Flows

Flow: <short title>

Goal: As a [developer/integrator/mesheryctl/operator] user,

I would like to …….. ,

so that ……..

Actors:

1.

Preconditions:

1.

Assumptions:

1.

Behavior:

1.

Implementation:

1.

Error Handling:

1.

Acceptance Criteria:

1.

Flow: <short title>

Goal: As a [developer/integrator/mesheryctl/operator] user,

I would like to …….. ,

so that ……..

Actors:

1.

Preconditions:

1.

Assumptions:

1.

Behavior:

1.

Implementation:

1.

Error Handling:

1.

Acceptance Criteria: