Asynchronous Service Communication

Problem Statement

Most modern web service architectures rely on asynchronous RESTful services. In certain instances, especially with ML architectures like Citrine, longer running complex tasks will make it difficult to fulfill promises in a timely fashion (i.e. < 500ms) therefore a more robust mechanism for managing asynchronous communication is required.

There are two primary types of asynchronous communication scenarios that this page attempts to address:

1. **Request-Response** : An example of this would be a user request requiring complex configuration. The validation of the inputted configuration may take longer to analyze (e.g. may need to get responses from more than one source) than a reasonable time-out period would allow.
2. ~~**Publish Subscribe**: A user may periodically want to check status of a longer running process. We also may want a place to put general status messages as part of a larger logging framework. These messages could be success, fail or provide completion status. We could also track user actions. This is especially important since most operations will be managed at the project level.~~
3. *Is there another use-case which could require another pattern*

Primary Goals for consideration

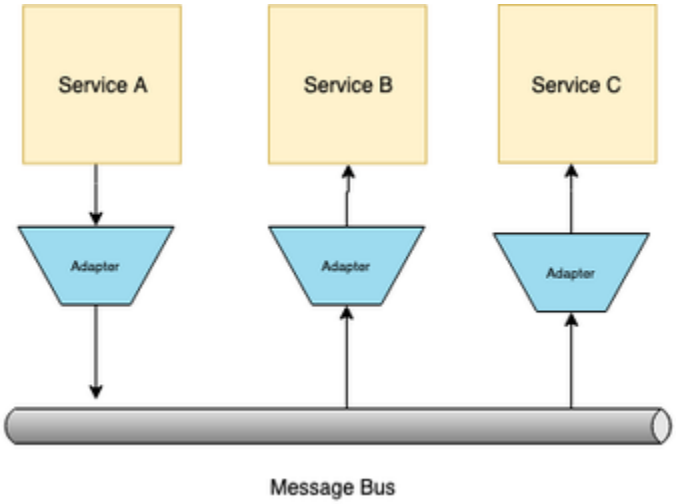
1. **Service Coupling** – Try to minimize dependencies between services. Services should be able to evolve without breaking each other
2. **Message Schema** – Services must agree on a common messaging schema We should consider that the schema needs to be extensible in order to support future evolution
3. **Reliability** – Since we're relying on this for communication between services, reliability has to be a primary (if not the primary) concern
4. *Any thing else?*

Non-Messaging Solutions

1. **File Transfer** – highly de-coupled but impractical but here for completeness. Using files generally introduces a fair amount of latency and potentially deadlocking and other race conditions. Should be limited to large data transfer.
2. **Shared Database –** we could define a schema that supported the message types we require (we'll have to do that anyway) and push messages to a shared DB (i.e. Dynamo or and RDBMS). The process for the message producer would be straightforward and message consumer could poll the database at regular intervals. This wouldn't be particularly elegant and while status messages would be fairly straight-forward (eg. log messages where we aren't tracking message consumption) messages where we would want confirmation they were consumed would introduce complexity (i.e we would need to update the db record or keep a separate table). A shared database solution should only really be considered as an alternative to service-polling for status (see below), since a database can at least be optimized for the process.
3. **Remote Procedure Invocation (or RPC) –** services could set up with specific end-points for messaging. In the case of a Request/Reply, the URI of the endpoint could be passed as a parameter in the request and once the result was generated, a message could be pushed to response endpoint (remote callback). This will certainly encapsulate the mechanics behind the production and consumption of the message but this is the most coupled solution presented here. Changes to either side (including perhaps extending the schema) need to be coordinated. As this is a stateful coupling (both sides making up a transaction), if the calling service is down there could be state inconsistencies. Additionally, this would require a non-RESTful endpoint to be added to the service. This type of communication doesn't really make sense for messaging, remote callbacks, like all RPC's should be used when additional functionality needs to be invoked remotely.
4. **Status Polling –** services could provide RESTful endpoints to check the status of an operation. Here the operation would be considered a resource so repeatedly calling a GET on that resource could provide the equivalent of a status message. Issues here include that the service maintaining the resource needs to persist that resource longer than otherwise necessary in order to support the status call. Also, the normal operation becomes making repeated requests and detecting for change, which is a wasteful use of resources. ~~If the calls are very frequent and return the same information, it logically makes sense to cache, perhaps introducing a CDN~~ 🙂.Much more efficient should be pushing the result to a database and ask the client retrieve it from there. Polling operations should be left to databases, they can be optimized for this sort of thing (i.e. read-replicas)
5. *Anything else we want to consider here?*

Messaging

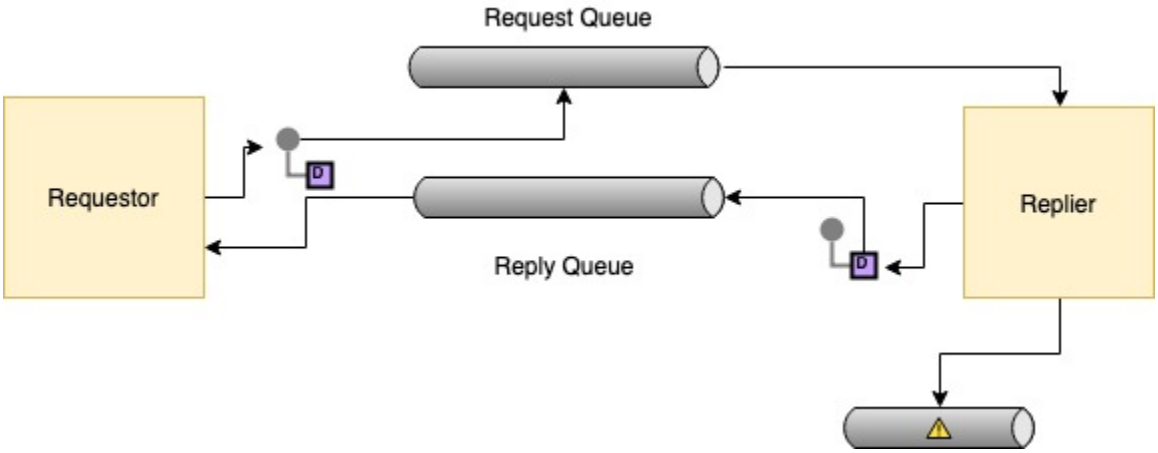
A messaging solution addresses the challenges of asynchronous communication without some of the drawbacks on non-messaging solutions. Like file transfer, a messaging solution allows services to be very decoupled but without obvious challenges involved with managing files. As mentioned above, file transfer should be limited to large chunks of data and we are concerning ourselves, hopefully, with smaller payloads. A shared database, like a messaging solution can keep all the data is an accessible place but doesn't give us enough flexibility in managing those messages, especially if we wanted to introduce more collaborative behavior in the future, it also potentially couples the services to the database although that could be abstracted out. Finally, a messaging solution can be used to build the same functional benefits of a RPC approach without the strong coupling and much better support for one-to-many type communication.



Messaging Patterns for Consideration

In the following sections I will be describing solutions in terms of the Enterprise Integration Patterns (EIP) they consist of. Patterns can be identified in **bold.** The patterns themselves were defined by Gregor Hohpe and Bobby Woolf in Enterprise Integration Patterns and have been summarized here for the most part. All of the Integration patterns defined below are widely implemented and are supported natively in ActiveMQ.

Request-Reply



The message should be fairly straight-forward. Include a timestamp, message id and the reply-to queue. For example:

Let's begin by looking at the canonical Request-Reply pattern. Typically, Request-Reply is used to send **Command Messages** however our use-case would be to send a **Document Message** (footnote here) so that's where we will focus. We will assume we need **Guaranteed Delivery** h owever . **Message Expiration** would not be a concern. The messages make up a transaction so we need to have ID's to correlate. At a minimum the Requestor Message should have a timestamp, Message ID, reply queue and the message. For example:

```
Sent request
 Time: 1552959300136
 Message ID: 61934db2-f9d7-47e2-be80-9706615aae06
 Correl ID: null
 Reply to: Reply Queue
 Contents: Citrine Rocks!
```

The response to this type of message would be fairly straight-forward since all we are looking for is an acknowledgment - this acknowledgement message then requires a reference to the original message and a timestamp. It should also include the contents of the original message for verification. For example:

```
Received request
 Time: 1553020348447
 Message ID: 62baf6697-20fe-4683-964a-b92779275291
 Correl ID: 61934db2-f9d7-47e2-be80-9706615aae06
 Reply to: null
 Contents: Citrine Rocks!
```

So the correlation ID is the original message ID and the reply address is blank (indicating that this is not a request message). The timestamp clearly shows that this message was created after the original and the Contents could be compared to the original message for verification.

Would want a response from the original caller to inform the receiver that the response was received?

For Citrine, this pattern could be useful. While we will not use Request-Reply by itself, let's discuss a possible hybrid. Let's consider making the Request a REST call rather than a message. The basic flow is identical with the exception of the initial endpoint.

```
POST /orion/workflow/create HTTP/1.1
Host: 9d50f8c3-e564-4d7c-923c-7d50e804b6b3.mock.pstmn.io
Content-Type: application/json
Time: 1552959300136
Message-ID: 61934db2-f9d7-47e2-be80-9706615aae06
Reply-to: Reply Queue
cache-control: no-cache
Postman-Token: 53337f22-367c-421e-82ba-d8ec84bc8917
{
  "Workflow": "46371a70b2cf3953b12b3e0bf8a83ea2",
  "Modules": [
    {
      "GridCapability": [
        "23c83140-05df-4b53-9ccf-ce1fcfc22773"
      ]
    }
  ]
}
```

So we can imagine a request like the one above and then, taking the place of the original Request message. The response message would remain the same:

```
Received request
 Time: 1553020348447
 Message ID: 62baf6697-20fe-4683-964a-b92779275291
 Correl ID: 61934db2-f9d7-47e2-be80-9706615aae06
 Reply to: null
 Contents: {result: 'Everything is awesome'}
```

DISCUSS!!!

Coming soon Publish-Subscribe