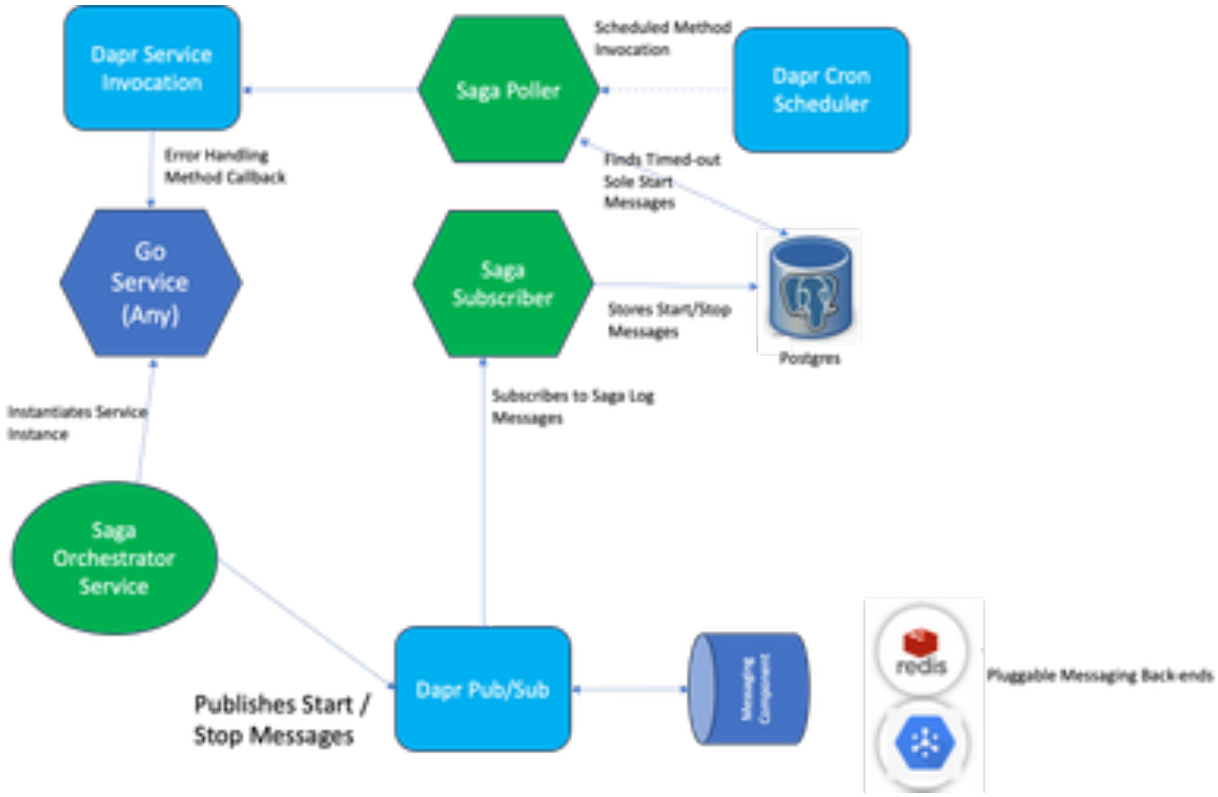
Saga Execution Orchestrator Example

Given the Saga pattern [Saga Pattern] this section details how this can easily be implemented using Dapr and the basic building blocks provided. This example project shows how capabilities such as monitoring are available from the framework.

The Solution Architecture is shown below:



There are 3 logical components of this solution:

1. The Saga Service Code which provides an interface and is instantiated into the Calling Go Service code
2. The Saga Subscriber
3. The Saga Poller

These components use Darp capabilities to reduce the amount of code required.

```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Go                               9            174             78            658
YAML                            12              7              1            339
Markdown                         1             25              0             83
Makefile                         4              1              0             25
-------------------------------------------------------------------------------
TOTAL                           26            207             79           1105
-------------------------------------------------------------------------------
```

The figures for GO code include a couple of Go test programs!

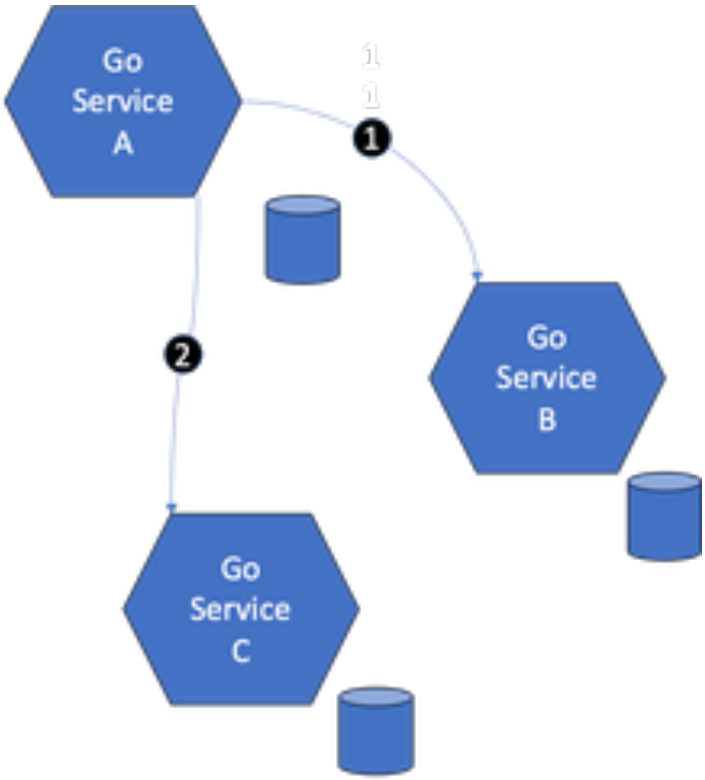
The client Go service code is linked with the Saga Service code. This provides methods for the client service to publish Start & Stop messages to a queue managed by Dapr. This ensure that the latency to the consuming Go service is minimal.

The Saga Subscriber component reads these messages and stores them in a database, the Saga Log, using a Go native Postgres driver. Originally, I used the Dapr DataStore, but this is process specific so I switched to Postgres. Only Start messages are stored and these are deleted when a Stop message is received.

The Saga Poll queries the State store for Start messages that exist and for which the timeout period has elapsed. When found the client's service call-back method recorded in the Start message is invoked. If successful the Start message in the state store is deleted to avoid a repeat of the call-back method.

Usage Scenarios

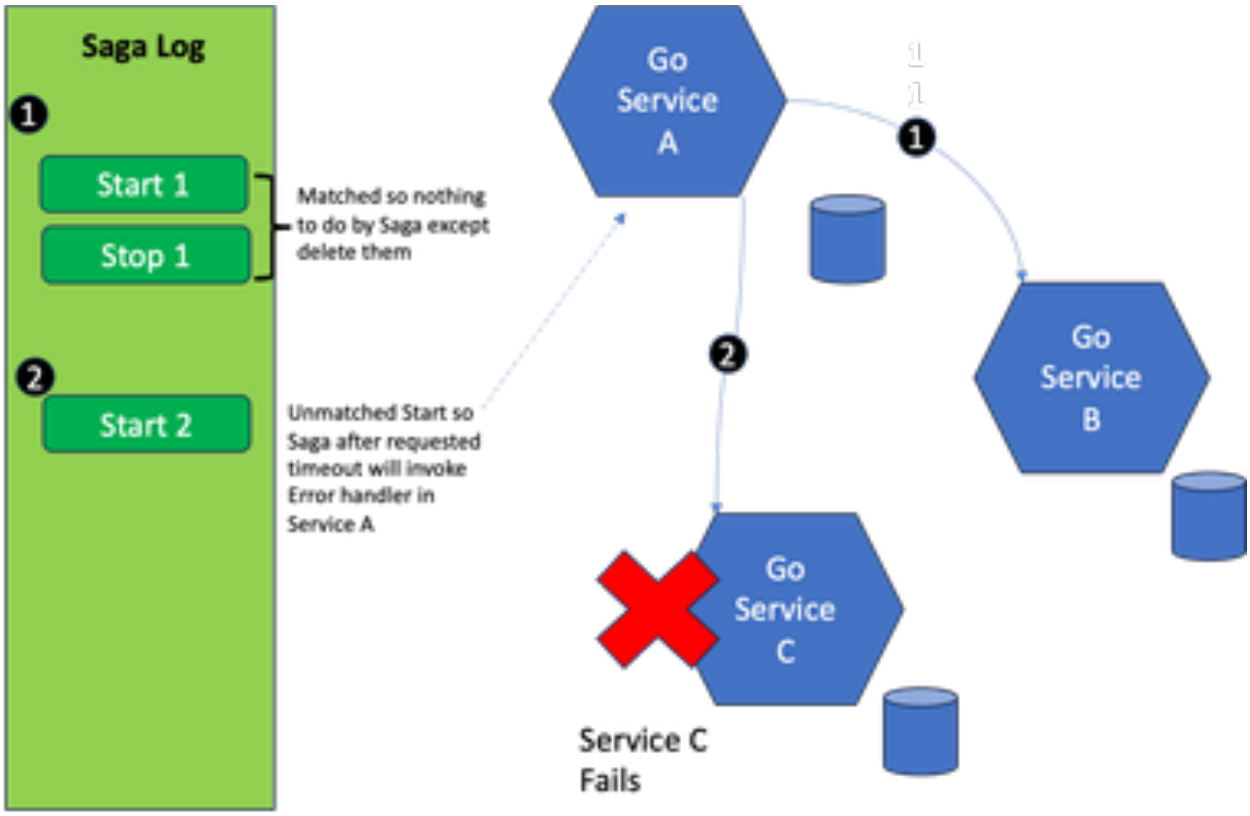
Assume a service is having to call two other services as part of a logical transaction:



Scenario 1 Happy Path

In this case everything is ok so the Stops logged by Service A will cancel out the Starts by the Saga Subscriber. There will be nothing to recover. The Saga Scheduler will ensure that the Saga Log is empty after Service A has successfully completed both calls.

Scenario 2 Service C fails



In this case something has gone wrong with Service C. Assuming retries have happened and Service C is still not responding, the the Saga Poller will detect that there is an unmatched Start 2 message and after the configured timeout it will call the error call-back handler passed in the Start 2 message. This message can contain json data in addition to a GUUID token based by Service A that will enable Service A to take the appropriate error recovery. This will be service specific, but could involve reversing the change made by invoking Service B again.

At the end of the recovery processing the Saga Log will be empty. The Start 2 message will remain in the Saga Log until he error handler method in Service A has been invoked.