community how-to resources

Aggregator Message Pattern with Hazelcast

by Hazelcast Community April 1, 2016 Comments Off

Kevin Denver from toastshaman.com recently published a blog post entitled, Aggregator Message Pattern with Hazelcas"t. In the post, Kevin talks about how to combine HTTP requests in a load balanced environment, with code samples included.

...or how to combine HTTP requests in a load balanced environment

Every so often you come across a technical challenge that really excites you and you can't think of anything else for days until you find an elegant solution to solve it. This was one of these occasions.

RESTful Interface on top of a Legacy Pricing Engine

We were building a microservice with a RESTful interface for sending quotes to a legacy pricing engine. If you tell the pricing engine what product you would like and a handful of parameters for tweaking the margins of the selected product, you'll be given a quote. The quote includes the product you've selected, the parameters you've given the engine and foremost a price for the selected product and parameter combination.

The pricing engine only understands and speaks a verbose form of XML. Writing a microservice that would do the translation between a simplified JSON document and XML made sense from a business point of view because it makes it easier for third parties to integrate with the pricing engine. And in the future, it will be easier to substitute the legacy pricing engine with a pricing engine from a different vendor.

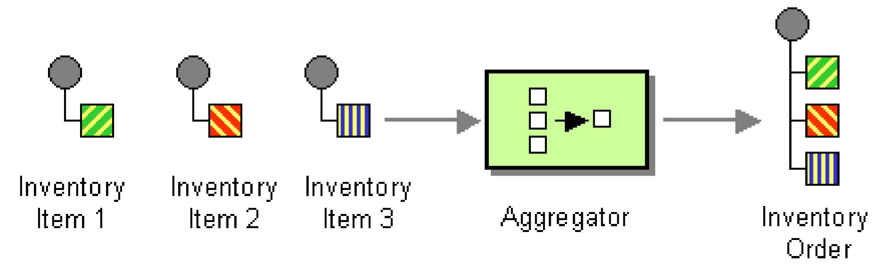
The pricing engine is nice enough to even give you a discount if you ask for more than one product. If you want to receive a discount you have to send a more knotty XML message to the pricing engine, outlining the various product and parameter combinations you've chosen. The third party consuming our JSON interface wasn't aware of the product discount capability of the pricing engine and therefore sent each product individually even though the customer might have selected a range of products. Obviously, we wanted to return the correct price to the customer that includes the discount. Asking the third party to change their one product per quote semantic wasn't an option at the time.

We needed to come up with a way of identifying whether a particular quote is actually part of a set of quotes that reward a discount. This shouldn't be such a difficult problem to solve if you had only one microservice receiving quotes. Doing this in a load balanced environment where you possibly have a vast number of instances of the microservice receiving individual quotes is trickier. Even more so because the request/response cycle is expected to be synchronous, hence we need to wait for all of the quotes in a set to arrive before we can send it to the pricing engine and ultimately return a response to the callees.

Aggregator Message Pattern

As it turns out, this is a common pattern as described in "Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions". The pattern we are looking for is called the Aggregator Message Pattern and is defined as:

Use a stateful filter, an Aggregator, to collect and store individual messages until a complete set of related messages has been received.



Sounds pretty easy! Apache Camel, the rule-based routing and mediation engine, for example, comes with an aggregator plugin out of the box. It seemed like the obvious choice but we failed to get it working. Aggregating the messages was easy enough but we struggled to notify all of the participants that the aggregation was successful.

A more low-level option is to use Hazelcast and implement the aggregator pattern ourselves. Hazelcast is an In-Memory Data Grid and ideal for sharing Java data structures across JVM boundaries. It supports standard Java Maps, Sets, Lists, Queues and even Topics. Hazelcast will allow you to store and share data across all instances in a clustered environment.

Hazelcast will share parts of your JVM's heap with other nodes in the cluster and hence the available memory will increase with each added node to the cluster. One of the main features of Hazelcast is not having a master node. Each node in the cluster is configured to be the same in terms of functionality. The oldest node (the first node created in the node cluster) manages the cluster members, i.e. automatically performs the data assignment to nodes. If the oldest node dies, the second oldest node will manage the cluster members. Hazelcast keeps the backup of each data entry on multiple nodes. On a node failure, the data is restored from the backup and the cluster will continue to operate without downtime.

The Aggregator Message Pattern with Hazelcast

I'll be using the term "message" instead of "quote" for the following section because the underlying principles apply to any data that is being aggregated, whether it is a quote or a bid on an auction. Also, in this scenario the "sender" is male (by tossing a coin) and relates to a third party system that is consuming the receiver's RESTful interface. The "receiver" is female and corresponds to any of the microservices in a load balanced environment that is capable of aggregating messages.

Let's quickly review on a high level what we're trying to achieve:

The sender sends single messages to the receiver and expects a synchronous response for each sent message. This means that the sender blocks the thread that started the conversation until a response is available. Unlike asynchronous messaging where the conversation is fire and forget and there's no need to wait for an immediate reply to continue processing. The receiver inspects an incoming message from the sender and determines whether it is part of a set of related messages that need to be collected and aggregated. The receiver applies an aggregation function to all of the messages that form a set, once all of them have been collected. The result of the aggregation is then returned to each conversation that the sender started.

For the receiver to determine whether a message is part of a set we needed to enhance the messages to include two additional fields: a transaction identifier and the number of expected messages. We count on the sender to provide these two additional parameters in each message he sends. Here's an example JSON document with the two fields appended:

```
{ transactionId: “B368B5C0-E6AE-11E5-BEA5-D181DC1050E5”, numberOfMessages: 5 }
```

Once the receiver collects the first message of a set, she initializes the Hazelcast-backed data structures she needs for completing the aggregation (Lines 10-15). Because we are using Hazelcast, the created data structures will be available to every instance that is connected to this particular Hazelcast cluster.

The collected messages will be stored in a multi value map where the key is the transaction identifier. A multi-value map is an appropriate data structure for storing and retrieving multiple values given a key. The map can be created upfront and reused between aggregations and there is no need to create a new one for each transaction.

In addition, she will create a Topic for signaling the success or the failure of the aggregation to all instances that received a message from the sender. The name of the topic will again be the transaction identifier. Topics can be used to distribute notifications to multiple subscribers (publish/subscribe) whereas a notification on a Queue can only be read by one subscriber at any given time. In our case, each instance will wait for the aggregation results to be published on this topic, before processing continues.

As you can see from the method signature, the main function returns a CompletableFuture (Line 18). A Future represents the result of an asynchronous computation and methods are provided to check if the computation is complete, to wait for its completion and to retrieve the result of the computation. The future acts as a lock and is only unlocked once the aggregation has either timed out, because not all of the messages were received in a configured time window, succeeded or failed (Line 15).

Once the data structures are all in place, the incoming messages can be stored in the multi value map (Line 21). This is being repeated until all of the expected number of messages have been received (Line 23). At this point, she can execute the aggregation function and provide the function with the list of all the received messages (Line 25). The instance that receives the last message of the set will execute the aggregation and publish the results on the previously created topic. By publishing a result on the topic (Line 26/28) we are signaling that the aggregation has either failed or succeeded and in turn complete the future and release the lock for all instances in the cluster simultaneously (Line 15). The lock ensures that every participating receiver waits until all messages have been collected and a result from the aggregation is available before returning it to the sender.

After the aggregation is complete, the data structures she created for this transaction will be destroyed (Line 39-41).

Final Words

Hazelcast proved to be well suited for orchestrating a number of microservices in a load balanced environment that do not share any resources other than the exposed data structures through Hazelcast.

When you configure your first Hazelcast cluster just be aware that by default it will use TCP Multicast messages to find other nodes in the network. This can lead to problems if your test environment shares the same network as your production environment. Nodes from the test environment and nodes from the production environment will share the same Hazelcast cluster. You can configure groups to avoid environments bleeding into each other.

```
 1  public class HazelcastAggregator<T, R> {
 2      private final String transactionId;
 3      private final int numberOfMessages;
 4      private final CompletableFuture<AggregatedResult<R>> onComplete;
 5      private final MultiMap<String, T> multiMap;
 6      private final ITopic<AggregatedResult<R>> topic;
 7      private final String messageListenerId;
 8
 9      public HazelcastAggregator(String transactionId, int numberOfMessages, HazelcastInstance hazelcast) {
10          this.transactionId = transactionId;
11          this.numberOfMessages = numberOfMessages;
12          this.multiMap = hazelcast.getMultiMap("aggregatorMap");
13          this.topic = hazelcast.getTopic(transactionId);
14          this.onComplete = new CompletableFuture<>();
15          this.messageListenerId = this.topic.addMessageListener(message -> this.onComplete.complete(message.getMessageObject()));
16      }
17
18      public CompletableFuture<AggregatedResult<R>> onMessage(T message, Function<List<T>, R> aggregator) {
19          try {
20              multiMap.lock(transactionId);
21              multiMap.put(transactionId, message);
22
23              if (numberOfMessages == multiMap.valueCount(transactionId)) {
24                  try {
25                      final R aggregatedResults = aggregator.apply(ImmutableList.<T>builder().addAll(multiMap.get(transactionId)).build());
26                      topic.publish(AggregatedResult.success(aggregatedResults));
27                  } catch (Exception e) {
28                      topic.publish(AggregatedResult.failed(e));
29                  } finally {
30                      teardown();
31                  }
32              }
33          } finally { multiMap.unlock(transactionId); }
34
35          return onComplete;
36      }
37
38      public void teardown() {
        multiMap.remove(transactionId);
        topic.removeMessageListener(messageListenerId);
        topic.destroy();
    }
}
```

```
@RestController
public class QuoteResource {

    private static final int TIME_OUT_IN_SECONDS = 10;
    private final HazelcastInstance hazelcast;

    @Autowired
    public QuoteResource(HazelcastInstance hazelcast) {
        this.hazelcast = hazelcast;
    }

    private static Function<List<Map<String, Object>>, Map<String, Object>> AGGREGATOR_FUNCTION = quotes -> {
        return ImmutableMap.of("message", format("I've aggregated %s messages", quotes.size()));
    };

    @RequestMapping(value = "quotes", method = POST)
    public Map<String, Object> quote(@RequestBody Map<String, Object> quote) {
        final String transactionId = getString(quote, "transactionId");
        final Integer numberOfMessages = getInteger(quote, "numberOfMessages");

        final Aggregator<Map<String, Object>, Map<String, Object>> aggregator = new Aggregator<>(transactionId, numberOfMessages, hazelcast);

        final CompletableFuture<AggregatedResult<Map<String, Object>>> response = aggregator.onMessage(quote, AGGREGATOR_FUNCTION);
        final AggregatedResult<Map<String, Object>> aggregatedResults;

        try {
            aggregatedResults = response.get(TIME_OUT_IN_SECONDS, SECONDS);
        } catch (Exception e) {
            aggregator.teardown();
            throw new IllegalStateException(format("Timed out waiting for all group messages %s %s", transactionId, e.getMessage()), e);
        }

        if (aggregatedResults.isSuccess()) {
            return aggregatedResults.get();
        }

        throw Throwables.propagate(aggregatedResults.getReason());
    }
}
```

← Hazelcast Python Client 0.2.1: A Getting Started Guide

Simulator 0.7 released! →