Protocol design in messaging architectures

Jan Macháček[‡]

May 10, 2018

Abstract

When a system needs to spread across asynchronous and unreliable communications boundary, its components on either side of this boundary have to use precise API. The protocol

Kafka is cool. Now, how to use it, run it, and what are the pitfalls.

1 State in distributed system

Imagine two systems that need to share information; a good starting point may be an endpoint where one system can request the state of the other system. If the first system needs to keep "in sync" with the second system, it needs to periodically query the endpoint to obtain the state from the second system. This keeps the two systems independent and decoupled, but it means that the systems might miss important state changes that happen between the two polls, and it makes both systems do needless work by querying the endpoints, even if the state remains the same.

1.1 Database as integration layer

This inefficiency, the complexity of building reliable and easy-to-use endpoints, the reasoning that both systems are views on the same underlying data, the possible time-pressures to deliver working integration, leads to the *database as integration layer* anti-pattern[1]. Never mind the *anti* in anti-pattern: what would happen if the systems actually decided to use a shared database? As the database spreads over multiple networked nodes, which introduces unreliability and delay, and means that the database administrators now need to choose the two of the C·A·P (C stands for consistency, A for availability, and P for partition-tolerance)–as long as one of the choices is P. This leaves the DBAs with a decision between A and C. A database that keeps its state on multiple nodes may keep them "in sync" by sending the nodes messages that describe the state transitions to be applied to the state. The state that the database (in the sense of all its nodes) keeps is then simply a reliable playback of all the messages. The state is a snapshot of the stream of messages at a given point in time. However, the decision between C and A still remains. If the choice is C, then all nodes have to agree that they have indeed received and applied all messages up to some point in time; in case the messages aren't received, or if there is no agreement, the database must become unavailable. If the choice is A, then there is the risk that a node that is being queried has not yet received all update messages; querying another node may yield different result; repeating the same query on the original node later may yield different result.

Using a database as an integration layer does not make the difficult choices of a distributed system go away, yet it keeps all the disadvantages of the anti-pattern. It forces shared data model, which may couple two (multiple!) systems together in the same release cycle; it is also a recipe for slowly-growing monolith, where one of the systems gains more and more functionality (the one with the largest number of developers); the other systems become lighter–after all, they are just views on the same shared state. The shared data model also means that two (multiple!) systems have to agree on usable data models for each.

1.2 Messaging to the rescue

If the systems that are to be integrated are event-driven–each system publishes messages as it proceeds with its operation, and if each system accepts messages at any time without previously initiating a request–then the integration can take advantage of this transfer of information. Relying entirely on [persistent] messaging can be burdensome; it is reasonable to rely on snapshots to remove the need to replay the messages from the beginning of time at start-up.

It might seem reasonable to rely on an endpoint that can deliver the entire snapshot in one response at start-up. Ignoring the potential problems with the size of the response, this will make the two systems to be integrated more coupled together. Specifically, it will remove the decoupling in time that the persistent messaging mechanism brings.

2 Implementation details

Consider two systems Ⓐ and Ⓑ that need to share parts of their state. If both systems are event-driven, then all that needs to be done is to Ⓐ in Figure 1.

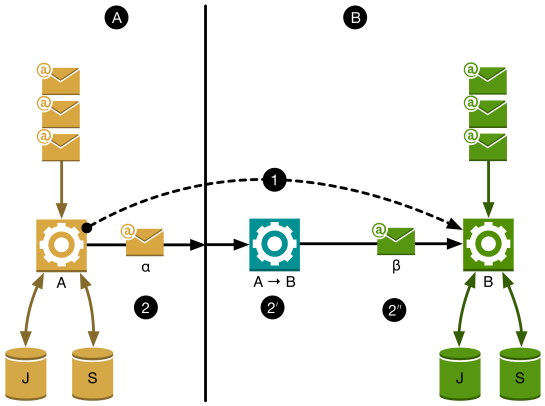

Figure 1: Integrating two systems

2.1 Custom code

System 1 <=> (Dynamo, Kafka) -> Output? => Input -> Kafka -> MySQL <=> System 2

2.2 AWS tooling

System 1 <=> Dynamo -> Lambda? -> Kinesis => Lambda -> MySQL <=> System 2

3 Practical applicaiton

X requested a feature in the Y system to implement a REST API endpoint that can be queried for the current state of a particular media assignment; but this request should also create a subscription for any changes to be delivered to the X system over a Kafka topic. The motivation for the feature was X's need to know Y's state, but to avoid doing periodic polling. (Polling, the team X reasoned, is just not cool, and might still miss an important change that happens between the two polls.)

- Y should not implement state that is only useful for X. The state Y would hold is not needed for its operation, it only serves one particular client;
- Maintaining subscriptions requires more state to be maintained inside Y; this state needs to be recoverable in case of failures, adding significant complexity to Y;
- Even with the subscription in place, the system (comprising Y and X) remains eventually consistent; adding arbitrary delays to the subscription does not solve this. (e.g. only start sending updates 40 seconds later to be sure that there was enough time for the state to become consistent does not solve the problem of eventual consistency; it simply allows us to pretend that the system is consistent!)
- There is no clear understanding (maybe there cannot be clear understanding) of when a subscription ends, or what identifies a client

Instead, I guided the teams to understand a better approach (bite the bullet and have X subscribe to the existing updates topic and maintain its own model, which duplicates information in principle, but allows most suitable representation of the information to exist in each system; it also maintains clear separation of responsibilities of each system; finally, it does not lock the two systems in the same release cycle).

References

[1] Bobby Woolf Gregor Hohpe. *Enterprise Integration Patterns*. An optional note. The address: Addison-Wesley, Oct. 2003. ISBN: 0321200683.