Protocol design in messaging architectures

Jan Macháček[*]

May 9, 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.

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.

Provisioning requested a feature in the BCP 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 Provisioning system over a Kafka topic. The motivation for the feature was Provisioning's need to know BCP's state, but to avoid doing periodic polling. (Polling, Provisioning reasoned, is just not cool, and might still miss an important change that happens between the two polls.) I offered architectural discussion; pointing out that: BCP should not implement state that is only useful for Provisioning. The state BCP would hold is not needed for its operation, it only serves one particular client; Maintaining subscriptions requires more state to be maintained inside BCP; this state needs to be recoverable in case of failures, adding significant complexity to BCP; Even with the subscription in place, the system (comprising BCP and Provisioning) 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 Provisioning 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).

I consider the architectural assistance to be valuable, but the task of the P.E. team only starts at this point. The P.E. team now needs to write and communicate the details of the advice and the decision; this communication needs to be lively (dare I say aspirational), taking the form of a paper, and example source code, and a blog post(s), and lunch-and-learn. Burying this content and advice in Wiki is not going to spread the knowledge as far as it needs to go.

2 Composability

References

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