Integrating microservices

Jan Macháček[‡]

June 18, 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 Stateful distributed systems

Systems whose components run on multiple nodes are difficult to design without obvious errors, difficult to build, difficult to run, and difficult to maintain; the reasons that justify this complexity are resilience and performance. Resilience refers to the system's ability to continue to operate even if some of its components are failing; informally, performance refers to the system's ability to handle large incoming workload quickly. The exact definitions for "large" and "quickly" <- TODO.

1.1 Why?

A system is spread over multiple nodes because it is too big for just one node, or because the risk of node failure taking down the entire system is unacceptable. The size of the benefit of distributing the system over mulitple noes depends on the system's workload and architecture. The best-case scenario is a system whose work units fit entirely on one node, and do not need to communicate with any other nodes. Consider a system that can perform independent image classification where the image to be classified fits into one input message, the classifier does not need any further input to perform the classification, and the result can be delivered in a single output message. Such a system is completely stateless and so very simple to distribute: the increase the system's throughput, one just needs to add nodes.

It might need to store too much data to fit on the built-in disks, it might need to do too much computation for the CPU to handle, it might need to keep too much data to keep in RAM, etc. This reasoning assumes that all these local resources (CPU, RAM, disk) are

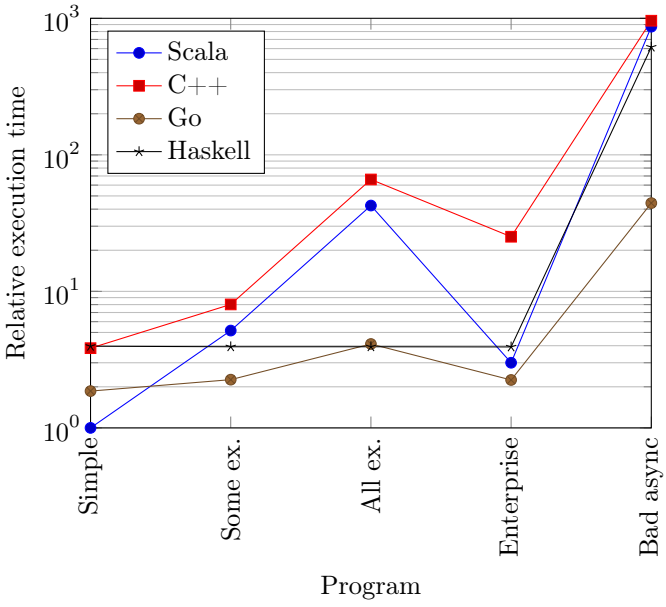


Figure 1: Simple computation

2 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[2]. 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.

3 Message-driven systems

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.

Consider two systems that need to share parts of their state. If both systems are event-driven, the task is simpler: all that needs to be done is to deliver the messages from the first system to the second system. Even though it is possible for the second system to accept the "internal" messages from the first system, it is safer to add a component that implements translation between the two systems. (This translation component can be $identity$, but it should be there.) Figure 3 illustrates the event-driven scenario.

Both systems Ⓐ and Ⓑ have their private journal and snapshot stores. The sharing of state begins by initial query that pulls in the appropriate snapshot of state from system Ⓐ into Ⓑ in ①; then the system Ⓑ follows the state changes in Ⓐ by acting on the [translated] messages it receives from Ⓐ. The system Ⓐ is not aware of any connection; it is not even aware of the translation component; all that the system Ⓐ is concerned about is delivery of messages to its messaging boundary. The flow is illustrated in ②, ③, and ④.

Figure 3 does not explain how the messages cross the boundary between the systems; it is clear that there needs to be some kind of connector that sits either on the Ⓐ or the Ⓑ side. It is better for the connector to be on the Ⓑ side. Ⓑ wants to receive messages from Ⓐ, so it should be responsible for making the arrangements: it should verify that it is indeed connecting to the *right* Ⓐ (checking certificates, for example); even more importantly, the connector needs to *consume* messages from Ⓐ, and then *publish* the same messages to Ⓑ. Publishing is potentially much more complex procedure (what is the acceptable confirmation / acknowledgement, what are the maximum batch sizes, timeouts, etc.), and should be in full control of Ⓑ[1].

All other approaches are recpies for problems (now or in the very near future). By far the worst-case scenario is database-to-database integration; particularly where Ⓐ writes directly to the database of Ⓑ. This is the most brittle and most dangerous approach; only slightly less dangerous and brittle solution is for Ⓑ to pull data from Ⓐ's database. If the system Ⓑ is not event-driven, the only approach is to poll data from Ⓐ[1].

If the two systems are connected "from the beginning of time" (that is Ⓑ receives all messages from Ⓐ from the first message), or if the two systems only need to synchronise a fixed number of messages, there will be no need to worry about the initial snapshot poll from Ⓐ. In other cases, Ⓑ requests the initial state from Ⓐ and then subscribes to the updates. Unfortunately, Ⓐ keeps updating its state, even as it serves the response to the initial poll to Ⓑ. A simple algorithm that Ⓐ can follow– $state \leftarrow poll(B);$ $subscribe(B);$ –runs the risk of missing messages that happened just after the initial poll. A systematic way of solving this is to include an offset in the initial poll response specifying the position at which the snapshot was taken. Ⓑ is responsible for subscribing to updates from Ⓐ from that offset. This way, there is no risk in missing updates from Ⓐ. An alternative is to measure the risk of missing an update vs. doing too much work and subscribing from the end of the topic less some arbitrary offset.

A typical event-driven system with persistent messaging implemented in Kafka typically uses multiple topics, each with multiple partitions. The offset is associated with each topic partition; the response to the initial poll request needs to to include a map of $(topic, partition) \rightarrow offset$. Figure 4 illustrates the message contents and

[1] To avoid the inefficiency of frequent polls with no changes, or too infrequent polls missing changes, add a component that can act as fancy circuit-breaker

flow.

The first message is the response to the request for initial state. It contains the state (in the *body* field) and the topic partition offsets used to construct the state. After processing the initial state Ⓑ subscribes to the topics indicated by Ⓐ to receive updates. This gurarantees that no messages will be lost, though it does not guarantee a *total ordering* of the update messages on the topic; Ⓑ has to be able to deal with out-of-order delivery of the update messages[2]. Listing 1 shows a typical code in Ⓑ.

```
if not initialised then
    (tps, state) ← poll(A)
    insert(state)
    subscribe(mode = SemiAutomatic, tpos = tpos,
              onMessage = λm → update(m))
else
    subscribe(mode = FullyAutomatic,
              onMessage = λm → update(m))
end if
```

Listing 1: Message flow

The ***not*** *initialised* indicates that Ⓑ has never been synchronised with A; the *mode = SemiAutomatic* instructs the messaging infrastructure to handle offset management, but allow initial offset specification; the *mode = FullyAutomatic* indicates that the messaging infrastructure should start the subscription from the last consumed offset.

4 Implementation

This paper will offer two implementation approaches: one that relies on Apache Kafka and one that uses the AWS tooling. Other implementations will find many common principles between the two.

4.1 Kafka-based

If the system cannot take advantage of any built-in functionality to the underlying infrastructure, it will be necessary to build the connectors and transformers ourselves.

System 1 <=> MySQL -> Kafka => Mirror Maker -> Kafka Connector -> Kafka -> Kafka Connector -> MySQL <=> System 2

4.2 AWS-based

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

5 Scalability and resilience

TODO.

[2] Distributed messaging systems such as Kafka[4] or AWS Kinesis[5] only have total order in a specific partition of a topic.

6 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

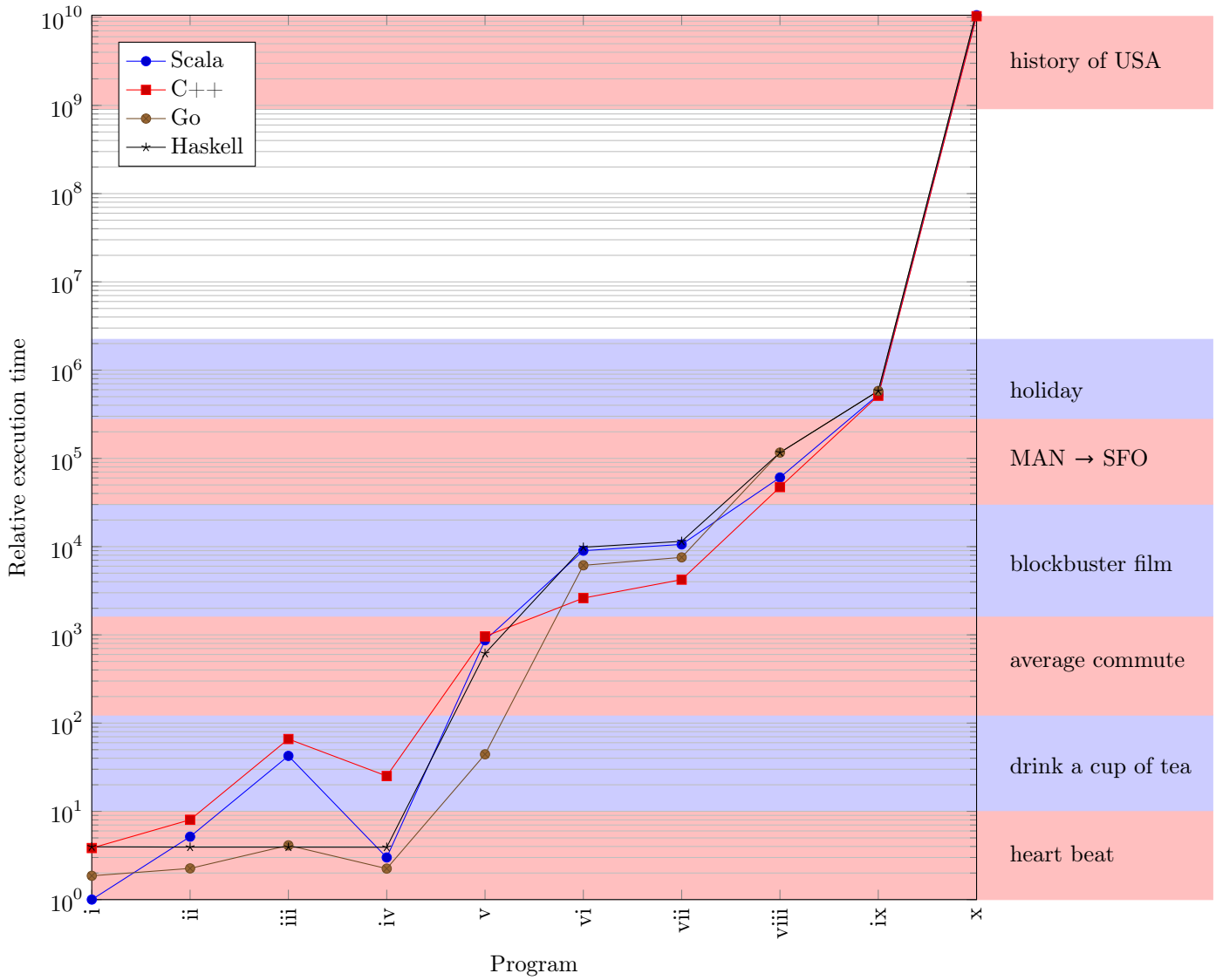
[1] Confluent. *Apache Kafka MirrorMaker*. `https://docs.confluent.io/current/multi-dc/mirrormaker.html`.

[2] Bobby Woolf Gregor Hohpe. *Enterprise Integration Patterns*. Addison-Wesley, Oct. 2003. ISBN: 0321200683.

[3] Sam Halliday. "High-performance linear algebra". In: Presented at Scala Exchange 2014, Dec. 2014.

[4] Apache Kafka. *Apache Kafka*. `https://kafka.apache.org`.

[5] AWS Kinesis. *AWS Kinesis*. `https://aws.amazon.com/kinesis/`.



i Simple

ii Some ex.

iii All ex.

iv Enterprise

v Bad async

vi 1 MiB from `/dev/zero`

vii 1 MiB from regular file

viii REST API call with 100 µs latency

ix REST API call with 10 ms latency

x REST API call with 0,000 01 error rate + 10 ms latency

Figure 2: Simple computation

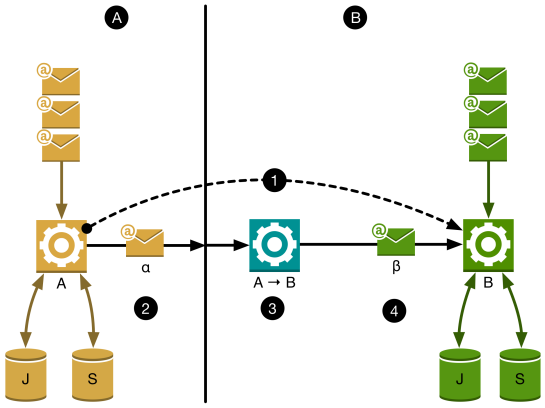


Figure 3: Integrating two systems

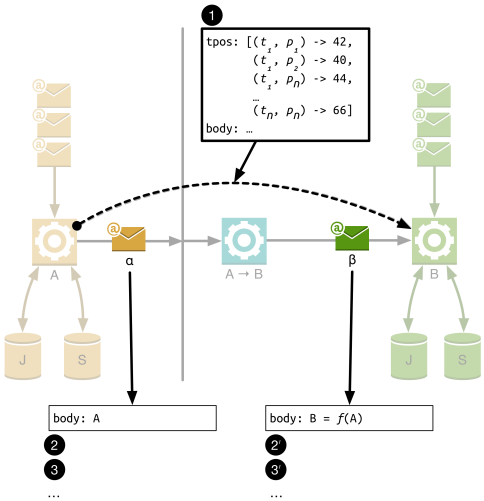


Figure 4: Messages