Integrating microservices

Jan Macháček[‡]

June 4, 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.

Table 1[3] shows typical latencies in a computer system translated to human timescales; it demonstrates that "the further the CPU has to go, the longer it takes".

XXX

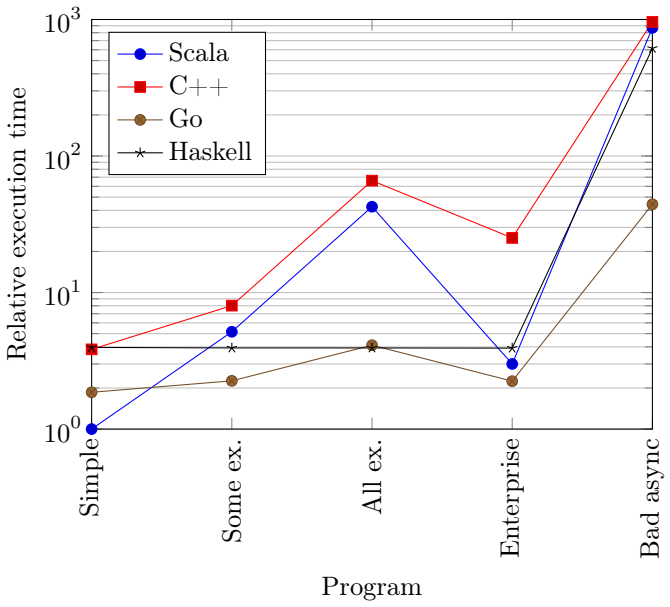


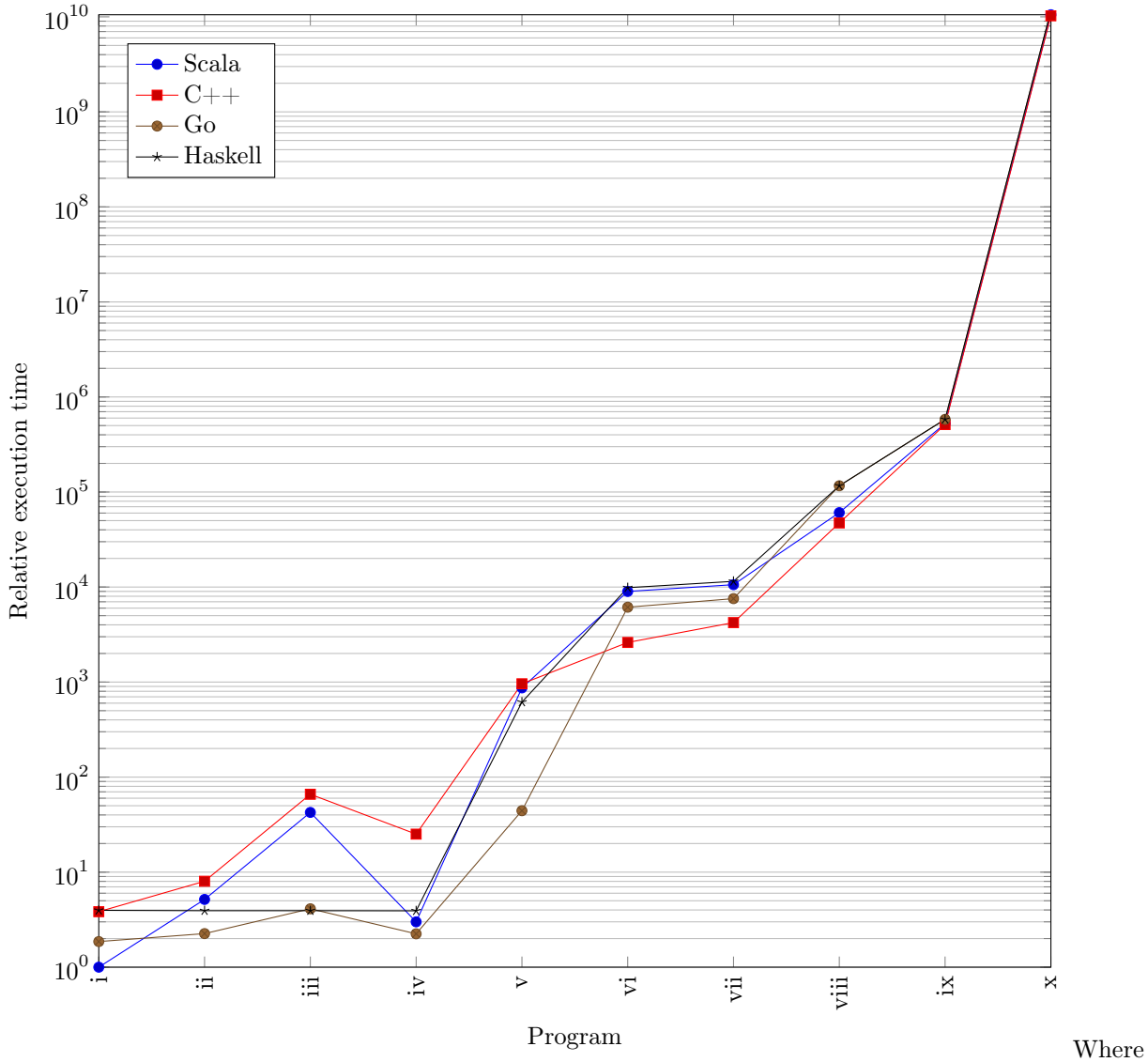
Figure 1: Simple computation

Computer	Human	Analogy
L1 cache reference	0.5 s	One heart beat
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Long yawn
Mutex lock/unlock	25 s	Making a cup of tea
Main memory reference	100 s	Brushing one's teeth
Compress 1 KiB with Zippy	50 min	Build Scala
Send 2 KiB over 1 Gbit s^{-1} network	5.5 h	London to Edinburgh
SSD random read	1.7 d	Weekend
Read 1 MiB sequentially from memory	2.9 d	Long weekend
Round trip within same datacentre	5.8 d	Short holiday
Read 1 MiB sequentially from SSD	11.6 d	Holiday
Disk seek	16.5 w	Term of university
Read 1 MiB sequentially from disk	7.8 m	Fully paid maternity in Norway
Send packet CA → Netherlands → CA	4.8 y	Government's term

Table 1. Low-level latency

Program	Implementation & Runtime				
	Scala	Go	C++	Haskell	Python
Simple 1.00	3.84	1.86	3.97	0	0
Some exceptions 5.16	8.02	2.26	3.93	0	0
All exceptions 42.51	66.00	4.12	3.93	0	0
Enterprise 3.00	25.12	2.24	3.92	0	0
Bad async 868.50	959.99	44.30	615.51	0	0

Table 2. Runtime



Where

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

]

2 State in distributed system

Imagine two systems that need to share information; a good

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

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 3 illustrates the message contents and 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 Ⓑ.

[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

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

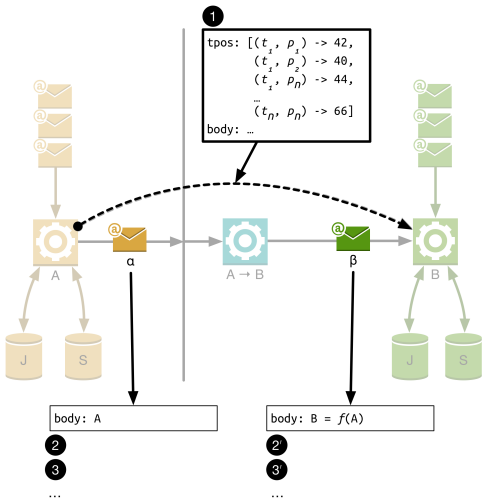


Figure 3: Messages

```
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.

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/`.