Consistency and Availability in Microservice Architectures

Davide Rossi(✉)

Department of Computer Science and Engineering,
University of Bologna, Bologna, Italy
rossi@cs.unibo.it

Abstract. For the most part, the first instances of microservice architectures have been deployed for the benefit of the so-called Internet-scale companies in contexts where availability is a critical concern. Their success in this context, along with their promise to be more agile than competing solutions in adapting to changing needs, soon attracted the interest of very diverse classes of business domains characterized by different priorities with respect to non-functional requirements. Microservices embraced this challenge, showing a unique ability to allow for a plethora of solutions, enabling developers to reach the trade-off between consistency and availability that better suits their needs. From a design point of view this translates into a vast solution space. While this can be perceived as an opportunity to enjoy greater freedom with respect to other architectural styles it also means that finding the best solution for the problem at hand can be complex and it is easier to incur in errors that can put a whole project at risk. In this paper we review some possible solutions to address common problems that arise when adopting microservices and we present strategies to address consistency and availability; we also discuss the impact these strategies have on the design space.

Keywords: Microservices architecture · Service-Oriented · Architecture · Software architecture

1 Introduction

All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change (Grady Booch as cited in [1]).

This citation ties together software architecture and design decisions. Implicitly it also ties together software architectures and non-functional requirements since it is obvious to anyone who has been involved in software development that the decisions for which the cost of change is higher are the ones made to address this class of requirements (think about improving the scalability of a system that has not been designed from the start to allow for that). In this respect we can say that non-functional requirements are the main drivers behind the design choices that shape a software architecture [2]. How design decisions and non-functional requirements play together in microservice architectures is the main topic of this paper.

M. J. Escalona et al. (Eds.): WEBIST 2018, LNBIP 372, pp. 39–55, 2019.
https://doi.org/10.1007/978-3-030-35330-8_3

Microservices architecture (or, simply, microservices) represent an architectural style.

Architectural styles are about constraints [3], which means that when an architectural style is adopted the design decision space is constrained.

Service-Oriented Architecture (SOA) represent an architectural style as well, a more generic one with respect to microservices in which the latter impose more stringent constraints on loose coupling, remarking that each service can be developed, deployed, and scaled independently, which is somehow related to the "products not projects" characteristic from the often cited list composed by Lewis and Fowler [4].

Other kinds of SOA exist, of course, one that is often compared to microservices is what in this paper is referred to as Enterprise SOA (E-SOA). The word enterprise here suggests we are addressing architectures designed to support non trivial non-functional qualities, since most enterprise software has to cope with consistency, availability, data integrity, robustness, security and so forth (notice that in this paper availability will often be used as an umbrella term encompassing related qualities such as performance and scalability, the same applies to consistency that encompasses also the likes of integrity and durability).

Most E-SOA solutions adopt some kind of support to ease many of the recurring problems that arise when building critical distributed systems so it is no wonder that many of these solutions are built on top of large platforms like JEE and .NET and adopt infrastructure software systems and middleware services, an example being the ubiquitous Enterprise Service Bus (ESB). Some of these solutions go as far as loading the ESB with too many concerns, even moving part of the business logic in it, a practice that created a bad reputation for a software component that, in some shape, is still needed in modern microservice architectures (this will be discussed more in depth later in this paper).

Using these combinations of platforms and infrastructure services implies that a set of architectural choices are already embodied in the environment hosting the application logic.

This approach is unusual for microservices-based solutions that leverage the large diffusion of enterprise-grade open source software proposing frameworks for various programming languages, data management systems (relational databases, graph databases, document databases, message brokers, ...) and infrastructure services and integrates them in various ways. As a result this opens up an array of choices when composing a microservices solution.

The CAP theorem [5] states that in a partitionable system it is not possible to achieve full consistency and maximum availability. Consistency and availability are in fact the most exemplary contrasting non-functional requirements that large, multi-user, distributed applications struggle with. In practical terms there is a price to pay in consistency to achieve better availability (for example by embracing relaxed consistency models like eventual consistency) and there is a price to pay in availability to achieve better consistency (just think about the contention caused by locking).

We can think about the trade-off between consistency and availability as a slider that moves between best consistency, no availability and best availability, no consistency. A peculiar characteristic of microservice is the ability to allow the slider to be moved in one direction or the other on a service-by-service or even request-by-request basis. This is something that is also possible with different approaches but at the expense of basic internal qualities such as simplicity, understandability and maintainability.

This paper presents a brief list of recipes that can be used in a microservices architecture to find the best balance between these two forces but also analyzes these recipes with respect to the impact they have on the design space. The dimensions of this space we are more interested in, in the context of this paper, are: governance, development, language (polyglot programming), data management (polyglot persistence) and platform/infrastructure. A detailed discussion of these dimensions is presented in Sect. 5.

We could argue that the array of choices allowed by microservices should not be intended as freedom that is here for the developers to take because of their personal preferences (a narration often supported in IT social media), rather as an opportunity to compose the right mix able to face the non-functional requirements needed by the application under development.

Which brings us back to the citation opening this section: architectural errors are the most costly ones, a project building on wrong architectural assumptions is hardly going to become a success story. Developers embracing microservices should be well aware of how their choices impact the non-functional qualities of their application and not be fascinated by IT social media articles.

This paper is structured as follows: in Sect. 2 a simple, yet paradigmatic and ubiquitous problem of microservices-based systems is introduced and a list of possible solutions to address the problem is presented. Section 3 contains an analysis of these solutions with respect to availability and consistency and a set of recipes to improve their ability to better address these concerns. Section 4 discusses relevant dimensions of the design space for microservices and how the aforementioned solutions impact them. Section 5 concludes the paper.

2 The Chain of Calls

Many aspects of a microservices architecture are impacted by non-functional requirements, however this paper focuses on a very simple issue that has the merit of being easy to understand, frequent to encounter and still triggering several of the pain points associated with many relevant design decisions. For each of these points, the best practices facing them will be presented along with a discussion on how these practices impact software qualities and design space.

This issue is here called the *chain of calls*. That name does not imply an actual cascade of invocations but refers to dealing with a request coming from a client that cannot be fully served by a single (micro)service in the system. From a conceptual point of view that means that service A needs a capability exposed by service B which, at its turn needs a capability exposed C and so on.

While this could very well happen with other architectural styles, the frequency of chains of calls is greatly magnified with microservices for the simple fact that they are *micro*, i.e. more focused on specific aspects of the domain so it is more likely that a single request needs the cooperation of multiple services to be served.

This is summarized by the following image, popularly known as the Microservices Death Star, a microservice dependency graph for Netflix's microservices as of 2012 (Fig. 1).

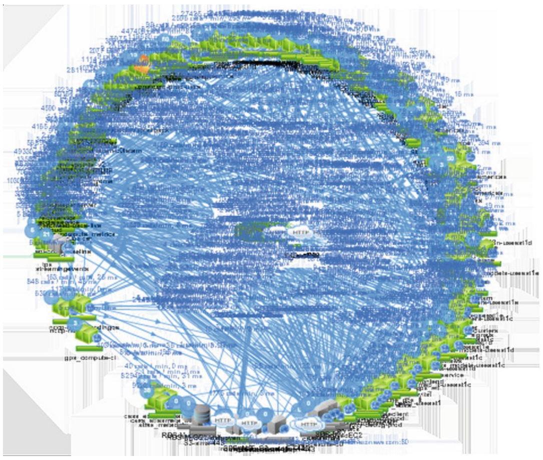

Fig. 1. The Microservices Death Star.

In E-SOA solutions most of the requests coming from clients are fully served by a single service and, for the rare cases in which a cooperation between multiple services is needed, most best practices suggest alternative ways of dealing with them (based on asynchronous messaging) instead of using a chain of calls (and we will see that these solutions work just as well for Microservices).

The section that follows presents some possible design alternatives that can be adopted. In the subsequent sections, following the rationale exposed in the introduction, these solutions are analyzed with respect to two different viewpoints: availability and consistency. A set of recipes to improve these solution's ability to better address these concerns is presented as well.

Running Example: When possible, a reference to the following elementary example will be used in this paper: an e-commerce application receives a request to retrieve information about a product including its description, price and whether it is available in stock (we can assume that a webpage for that product has to be presented to a user). Among the various microservices presented in the system are the `products` microservice (dealing with the domain of products: their description, their price, ...) and `the inventory` microservice (dealing with stock management).

2.1 Chain of Calls: Design Alternatives

As previously discussed, the chain of calls describes a set of cascading logical dependencies between microservices. This does not necessarily turn into an actual sequence of direct invocations. In fact, several strategies can be adopted to implement a chain of calls.

Here we distinguish between two main approaches: one in which actual invocations are performed and one in which the interactions between dependent services are decoupled (usually by using an asynchronous messaging infrastructure).

When actual invocations are performed we can further distinguish between choreography-based solutions and orchestration-based solutions.

Consider the example introduced in the previous section. In a choreography-based scenario, the external request is routed (usually by an API Gateway) to a microservice, possibly `products` since it has access to most of the information that has to be returned, then `products` invokes `inventory` to retrieve stock availability information, packs all the data in a response and returns it to the external client (via the gateway).

This simple scenario can be expanded at will: service A calls service B that calls service C that, depending on some logic decides to either call service D or service E and so forth. This is a choreography: it defines a coordination process between peers in the form of (observable) message exchanges. Each peer is responsible for generating the correct messages depending on the current state of the process.

Orchestration-based solutions, instead, make use of an additional component: an orchestrator that acts as a communication hub managing the interactions between services.

Digression 1, in the Appendix contains a discussion on configurable orchestrators.

In our example the external request is routed to the orchestrator that calls `products` to retrieve the product-related information, then calls `inventory` to retrieve the stock-related information for that product, packs all the data in a response and returns it to the external client.

Let us now see what options are available when using messaging-based solutions.

A very naive approach is to use asynchronous messages, possibly via a message brokering infrastructure, to decouple requests and responses from both a spatial (and possibly also a temporal) perspective: direct invocations are transformed in the emission of command messages from the caller and the emission of corresponding response messages from the callee. Service providers consume command messages while consumers consume response messages.

A peer-to-peer or an orchestrator-based approach can be adopted in this case as well, with the obvious additional indirection caused by the messaging infrastructure.

These solutions, however, are just removing the physical coupling while fully maintaining the logical one: the use of command messages in our example turns out to be not much different with respect to the naive approach previously discussed: when `prod-ucts` receives a request it creates a command message asking for stock availability, `inventory` listen di this message are creates a reply message that is then consumed by `products`.

Digression 2, in the Appendix contains a discussion on messaging and coupling.

More articulated solutions based on asynchronous messaging exist, while they have been around for many years now it is with the advent of domain-driven design (DDD) [8] that they found a conceptual framing. In DDD bounded contexts are used to separate the conceptual areas of an application domain; bounded contexts are then usually refined into the main components of the resulting application architecture, since DDD suggests that systems should be organized in a way that reflects the conceptual structure of the domain.

One of the possible ways to enable integration between these components is that of using asynchronous messaging in the form of events and commands, specifically domain events signal relevant occurrence in a domain whereas command messages are requests targeted to a domain.

Bounded contexts are not refined into microservices (although it is easy to read someone affirming the opposite, which is obviously wrong because of a granularity mismatch) but this integration mechanism naturally fits microservice architectures.

Let us consider our example again: each time a stock availability value changes in the `inventory` database, a domain event is published; `products` can, by listening to these events, keep a local copy of the availability information that is synchronized with that of `inventory`. With this approach the external request can be fully served by `products` and the chain of calls is actually avoided. This is not always possible, for example when service A needs a specific business function from service B, careful design of microservices and related bounded contexts should however limit this eventuality. This is a well-known approach in the E-SOA community and is gaining adoption in the microservices community as well.

To summarize, here are the available options to implement a chain of calls:

- CC1. Perform direct invocation
  - CC1.1. Use a choreography-based approach
  - CC1.2. Use an orchestration-based approach
- CC2. Use messaging
  - CC2.1. Use a choreography-based or an orchestration-based approach
  - CC2.2. Use a DDD-inspired solution and actually avoid chaining microservices

3 Chain of Calls: Analysis with Respect to Availability and Consistency

We now analyze the impact of the solutions presented in the previous section with respect to availability and consistency.

From an intuitive point of view the aim of this section is to show how the quality slider moves when adopting a specific solution.

We also present known strategies that are usually adopted to improve the limited quality that naive implementations can express with respect to consistency and availability.

Our analysis starts with CC1 (we collapse CC1.1 and CC1.2 here since we discuss overlapping concerns).

In CC1 direct invocations (synchronous calls) between microservices are performed. From an availability point of view the impact of this solution is easily recognizable: the external request can be served only if all the services involved in the chain are available (for simplicity here we assume that no fallback policies are available): if the average chain size is N and the average availability of each service is A, the overall availability of the system cannot be more than N^A, being N minor than 1 this obviously means that the system is less available than its services. For example: if the average availability for the services is 99.999% (also known as five-nines, a measure usually perceived as very good for a real-world system) and the average chain length is 5, the resulting availability will be 99.995%. That means an increase in downtime from 5 min 15 s per year to 26 min 17 s per year (which, for some classes of applications, could be unacceptable).

This very preliminary aspect, however, is largely overshadowed by a considerably more serious one: what happens in the presence of failures/delays.

It is well known that in an IP-based network a crashed process is indistinguishable from a slow one [6], in this context this means that when a response is not received after sending a request to a service that is part of a chain, there is no way to know if a response will eventually arrive or if the called service has crashed. To avoid for requests to be pending indefinitely, the usual approach is to assume a failure after a timeout. The duration of a timeout is usually determined with an heuristic taking into account the trade-off between the risk of considering crashed a service that is actually running (and maybe just experiencing a transient issue) or that of delaying for a long time a request that has no hope to be fulfilled (with obvious negative consequences on availability). The presence of a chain of calls exacerbates the problem of setting a reasonable time out since the slowness of a service impacts all the services that precede it in the chain, so perfectly healthy services can be assumed as crashed only because they are stuck waiting for their dependencies to produce a reply. The current best practice for microservice architectures (which usually employ some kind of virtualized infrastructure), stemming from the empirical observation that most invocation issues are due to transient problems (topology reconfigurations, virtual machine migrations, containers' virtual network modifications, garbage collection, etc.), is to set relatively short timeouts and perform retries.

Notice that retries are acceptable only when a system is designed to handle them, that usually means that services that are subject to retries should be idempotent: multiple invocations of the same request must lead to the same result; this can be achieved by designing requests to respect this semantic (do not allow requests like "decrement bank account by 10" but only requests like "set bank account to 1234", but then the service is exposed to unordered delivery issues) or by using some de-duplication mechanisms for incoming requests. The practice of setting relatively short timeouts and perform retries usually goes hand in hand with another practice that says that a service should be terminated as soon as it starts showing signs of erratic/slow behavior and replaced by existing replicas or newly created instances, an option that has been made possible by modern virtualized infrastructures and containers-based solutions in which the cost (and the time) of creating new service instances is minimized. To implement this approach, a health monitoring infrastructure has to be put in place, the infrastructure should gather health information from the services and interact with the network and the virtualization infrastructure to deal with the rerouting of messages to other services and failed services re-instantiation.

Since most invocation errors are due to transient issues, a simple retry usually solves the problem. There is, however, a minor but not insignificant number of cases in which the timeout is due to a service that is slowing down but still has not been identified by the health monitoring infrastructure and thus terminated. It is very well possible that the service is in a recoverable state and that the slowdown is due to transient overloading, swapping, garbage collection or similar issues. In those cases, however, retries are equivalent to punching a boxer trying to get back to his feet: the amount of requests arises, the service tries to fulfill them and slows further down, because of that the clients enter a timeout-retry loop until the service eventually fails under the overwhelming load. This could easily start a cascading failure effect that propagates to most (otherwise perfectly healthy) services in the system.

Basic mitigations include the use an exponential backoff algorithm to continually increase the delay between retries until the maximum limit is reached and back-pressure measures: when a service is on the verge of being overloaded it starts rejecting requests and sends failure responses signaling that the failure is not due to an error but to overload (however this requires cooperation from the calling services that have to delay their request even further or direct them to other replicas).

Circuit breaker [7] is a pattern vastly employed to improve stability and resiliency in microservice architectures in the presence of direct service-to-service invocation.

A circuit breaker acts as a proxy for operations that might fail. The proxy should monitor the number of recent failures that have occurred, and use this information to decide whether to allow the operation to proceed, or return a failure immediately.

The behavior of the proxy can be easily described as a state machine that can be closed, open or half-open. Details can be found in the aforementioned reference.

Another problem that can arise when dealing with multiple microservices calling each other is related to resources management. Shared resources (such as connection pool, memory, and CPU) when allocated to troubling connections (that suffers from long response times or are engaged in a retry loop) risk to starve other concurrent workloads. Bulkhead [7] is a pattern that suggests to partition service instances into different groups, based on consumer load and availability requirements (so, for example, a specific connection pool is used when communicating with a specific service, instead of using a single shared connection pool).

All these mitigations are usually mixed and require that all services in the system adopt the same policies with respect to them (imagine what could result if some services perform retries while others do not, only some adopt circuit breakers and so forth) so this has a huge impact in terms of governance.

Since it is not reasonable that all microservices deal independently with these recurring issues (otherwise most of the code will be filled with timeouts and retries instead of focusing on business logic) the usual solution is to move all the mentioned mitigations outside of the main code. This can happen with an in-process or with an out-of-process approach.

With the in-process approach a library is used to deal with service-to-service communications. A notable example is Netflix's Hystrix[1] that mixes the circuit breaker and the bulkhead pattern (and, indirectly, retries) but there are many others (e.g. Twitter's Finagle[2]). Of course a project could decide to implement its own library.

With the out-of-process approach an external, but colocated, proxy is used. The sidecar pattern [8] uses this approach, the sidecar usually also takes care of logging, monitoring and configuration issues which is pretty natural when we realize that all the requests are routed through this component.

The disciplined, consistent use of the sidecar pattern is at the roots of what is called a *service mesh* which is defined as a dedicated infrastructure layer for handling service-to-service communication [9]. A service mesh usually needs a lightweight virtualization infrastructure (i.e. containers) and a virtualization orchestrator (like Kubernetes[3]) to be deployed. This obviously results in stringent constraints associated to infrastructural choices.

Whether a system really needs all of these mitigation strategies mostly depends on the quality of service requirements that are imposed. It is important, however, to stress that software engineers should always have full command on the trade-offs between availability, constraints relaxation, and complexity of the systems. This means they should be aware of which solutions can be adopted and understand their impact on the overall architecture (which includes several limitations to the design space).

We now analyze how consistency is addressed when the chain of calls turns into a sequence of direct invocations. In this case, in general, when the involved services modify data, we are dealing with a distributed transaction. The usual solution to address consistency in distributed transactions is the adoption of mechanism based on the two-phase commit protocol. However, as the data management needs of Web 2.0 companies shifted the focus from SQL and ACID to NoSQL and BASE [13], the microservices community is more interested in trade-offs in which a price is paid in terms of consistency in order to achieve better availability. Two-phase commit is thus reserved to a very limited number of critical requests (if any) whereas most of the requests are served with relaxed consistency. Notice that two-phase commit is also very rarely adopted in E-SOA too, where messaging-based solution are usually preferred.

In order to guarantee some degree of consistency, microservices-based solutions, for the most part, adopted ad hoc solutions. These are colloquially known as feral concurrency control [15], that is application-level mechanisms for maintaining data integrity. At least this has been the case since recently, before finally realize that what has been done for twenty years now with E-SOA, WS-BEL and BPMN was often a viable option: explicitly identify choreographies/orchestrations and adopt long running compensating transactions (LTRs, which have now being re-popularized under the Distributed SAGA name in the microservices community, which is slightly inappropriate since in the original proposal [16] a SAGA has specific characteristics associated to interleaving). The basic idea is to define a mechanism to reach a relaxed form of atomicity by compensating the already executed steps of a transaction when the transaction itself fails.

[1] https://github.com/Netflix/Hystrix.
[2] https://twitter.github.io/finagle/.
[3] https://kubernetes.io/.

A long running transaction can make use of a coordinator (orchestration approach, which would be a natural mapping for CC1.1) or use a choreography approach (like in CC1.2). This second option, however, can result in some very complex issues that have to be dealt with: the state of the transaction is now a distributed state, in case of failures we must ensure its consistency (something that can be achieved using a robust distributed logging infrastructure). That also means we have problems with visibility and monitoring. This complexity usually leads to the adoption of orchestration-based solutions when consistency is a concern. In order not to compromise the reliability and availability of the system, the orchestrator, usually called the coordinator in this context, should not be a single point of failure and should be highly available (which adds to the overall complexity of the system).

See Digression 3 in the appendix for a discussion on configurable orchestrators and compensating transactions.

To summarize: the CC1 solutions needs quite a lot of effort to address high availability, mainly in the form of mitigation strategies associated to the issues related to the fail fast/retry policies. When this is done, microservice architectures have shown to be able to achieve very high levels of availability when adopting this kind of solutions (this is, for example, the case of Netflix).

On the consistency side, things are more blurred: strong consistency is expensive and is reserved for a limited number of critical requests; a relaxed form of atomicity is achievable by using long-running transactions (but with costs that are usually too high to justify when adopting CC1.1).

We now put CC2 under our microscope: these are solution based on asynchronous messages. The analysis of CC2.1 is quite straightforward: this is a solution of limited applicability since it does not improve significantly over its synchronous choreography or orchestration-based counterparts but it does add significant complexity, more so in the choreography case, which really makes the orchestration-based approach the only viable solution. In this setup the orchestrator becomes an asynchronous message coordinator and, besides the obvious considerations related to this fact, the analysis presented for CC1.2, from both a consistency and an availability perspectives, holds here too.

Much more interesting is the case of CC2.2. To better focus the problems raised by this solution let us get back to the e-commerce example: the adoption of CC2.2 in this case corresponds to implementing the `inventory` microservice in such a way that, when an availability update is persisted in its local database, a domain event is contextually produced. The `products` microservice listens for these events and updates its own copy of the stock availability accordingly.

With no further measures, this results in a system with no consistency guarantees of any kind: if the `inventory` microservice crashes after updating the database but before producing the domain event, the copy in `products` will not be reconciled.

Notice that this may very well be fully acceptable. A one-in-a-million error related to stock availability for a B2C e-commerce site can be just fine. But the same could not apply to a B2B site used by hospitals to acquire life-saving medicines.

Strong consistency, in this scenario, requires that whenever a domain event related to the modification of some information is generated, the persisting of this modification in the originating microservice has to be part of a distributed transaction in which the persisting of the local copies in all interested microservices participate. In general that turns into a distributed transaction (the domain of two-phase commit), which would impact availability (specifically performances and scalability) so severely to restrict strong consistency to a very limited subset of selected operations. The highest level of consistency for general operations in this scenario, in fact, is usually *eventual consistency* which means that is assumed that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value [14].

While opting for a relaxed form of consistency can be perceived as just adding a little more complexity, things can be more convoluted than that: to guarantee eventual consistency the database update and the generation of the domain event in the `inventory` microservice have to be atomic. There are a few solutions to achieve this, the easier one is to let the local database and the message queue participate in a multi-party atomic transaction (which is not necessarily a distributed one because they can both be local to the node hosting the `inventory` microservice). This, however, requires that the message broker supports atomic transactions, and the same applies to the database.

Enterprise-proof solutions to manage asynchronous messaging with transactional support have been around for a long time, they usually take the form of products presenting themselves as message queues or message brokers. But a new class of messaging management solutions is on the rise, an evolution that is similar TO the affirming of NoSQL and BASE in the persistence management domain. An example of the new class of messaging platforms is Apache Kafka [10], while other similar solutions are available we will mainly refer to Kafka as a paradigmatic instance of this new class. Kafka is presented as a distributed streaming solution, what makes this different from usual message queuing systems is persistence: events that are produced before a consumer is registered can still be retrieved. The main design goal in Kafka is clearly scalability but its wide adoption and its ease of use (along with its low cost, being an Open Source software solution) is extending its application domain to areas characterized by a large amount of events to process but also by stringent consistency requirements (like financial applications). Stringent consistency, however, usually implies integrity and atomicity and, while Kafka does support transactions, these transactions can be used to guarantee an exactly once delivery semantic but neither integrity nor atomicity (recovery logs are written asynchronously and particular failure patterns can lead to data loss).

Similar considerations can be extended to most NoSQL databases: ACID transaction are usually not supported which means that most combinations of messaging/persistence solutions adopted in microservices does not allow multi-party atomic transactions.

Ad hoc solutions to guarantee atomicity without multi-party atomic transactions do exist but are complex, brittle, need message deduplication support from listeners and, of course, still need some kind of transactional support from the database and/or the message broker. They are essentially an instance of feral concurrency control: for example the database can be used to record the produced domain events in the same transaction in which the update is performed (allowing the implementation of a form of checkpointing) or, conversely, the message queue can be used to store updates and let the service itself update the local database by consuming the same change events it produced (notice however that these are no more domain events, since domain events refer to something that has already taken place in the domain, which breaks some of the basic semantic assumptions in DDD).

From an availability point of view CC2.2 can be seen as an improvement over CC1 solutions under most real-world circumstances: to refer to our example if read requests are more frequent that write requests (i.e. users access product pages more often than they finalize purchases) most request can be served by only interesting the `product` microservice at the expenses of an event being generated by `inventory` (and processed by `product`) at purchase time. Asynchronous messages also promote decoupling between microservices easing the implementation of tailor-made scalability policies.

To summarize: CC2.2 is possibly the best option to meet stringent availability constraints (at the expense of an added complexity due to the managing of an asynchronous messaging infrastructure) but addressing consistency can be a problem: strong consistency can be an option only when serving a minority of the received requests and even eventual consistency adds relevant complexity and poses a relevant number of constraints with respect to the choice of messaging and persistence management infrastructure.

An interesting point that deserves to be emphasized is that an application built on microservices does not need to choose one of the CC solutions we presented but can mix them together and decide to adopt different consistency models depending on the specific task at hand, for example an e-commerce application can use a best effort approach for stock availability (e.g. CC2.2 with no consistency guarantee) until the user decides to finalize a transaction in which case strong consistency is used (e.g. CC1.1 with transactional guarantees) to verify the actual availability of the products.

It is also interesting to note that, as previously discussed, not all requests can be served with relaxed consistency, even in contexts with a large amount of potential clients (and thus with strong availability concerns). This is the case for domains like stock trading, gambling, micropayments and so forth. A very active stream of research focuses on this challenge, for example improving the availability of databases combining the ACID and the BASE models with modular concurrency control, like the SALT proposal [11].

4 Design Space Dimensions and Solutions Impact

In this section we detail some of the axis related to the design decision space that characterize the adoption of a microservices architecture and analyze which is the impact they have on the solutions presented in the previous sections. Of course there are other dimensions that could be discussed but are not presented in this paper, the collection we propose is mainly based on characteristics that are usually depicted as characterizing for microservices.

Governance is about policies. Policies are pervasive and can touch almost every aspect of software development and operations. E-SOA projects are usually characterized by a strong governance in which several aspects of the services (from both a design-time and a run-time point of view) are asserted and enforced. A bad management of governance, focusing only on collecting the larger possible set of policies, can actively inhibit change. Unfortunately far too many large E-SOA projects suffered from this problem. Microservices bring the promise of decentralized governance: centralized governance is perceived as an overhead that should be avoided by supporting service-specific governance and intra-service contracts (which can be promoted by using patterns like *tolerant reader* and *consumer-driven contracts*).

Development: microservices and agile programming have always been tightly coupled. The main reason behind that traces to the fact that to minimize the coupling between microservices they are usually developed as separate products. That translates to separate development projects and it is not unusual to have tenths if not hundreds of microservices in a single system. That calls for development methods with minimal overhead and agile programming is undeniably the best option.

In this respect, when we discuss development freedom we do not intend the freedom to choose between agile or structured approaches but the freedom to adopt different practices within an agile context. Most notably these practices could change between the projects of different microservices within the same application.

In this respect, then, the development category here is just a subset of governance. But since it receives significant interests from the microservices community it is presented separately.

Language: polyglot programming [6] has always been a strong selling point for microservice architectures. Since each microservice is a separate product, it can be developed with the language perceived as the most fitting to solve the specific problems that microservice has to address. This could easily result in an application developed with an array of different languages.

Data Management: just like polyglot programming, polyglot persistence [7] too has always been linked with microservices. A basic characteristics in SOA is that services should be autonomous and thus should take care of their own data. This is reflected in microservices at the conceptual level, where each microservice defines its own data model, but also at the implementation level, where it has the opportunity to select the most appropriate data storage solution.

Platform/Infrastructure: JEE and .NET provide well-defined ecosystems composed by libraries, frameworks and infrastructure services. Microservices can choose à la carte. An array of options is available, which is also possible thanks to the wide diffusion of enterprise-grade open source software.

It is easy to realize that most of the solutions (and mitigations) proposed in the previous sections have a large impact on these design dimensions. What follow is a brief analysis on what this impact is on a dimension by dimension basis.

Decentralized governance is affected by the adoption of policies to be applied to all (or most of the) microservices. This is for example the case of the fail fast/retry practice presented when analyzing CC1.1. We already discussed that it is not reasonable to think that different microservices use different strategies in this respect. When adopting the in-process solution this also immediately affects language: when libraries are used the languages to develop the microservices can only be the ones the libraries support. Human factors affect the choice of languages too: it is not unusual for a microservices-based application to be composed by tens of different microservices. Each microservice has its own development project with its development team. With disjoint teams this could imply a number of developers that only the likes of Amazon and Google can afford. In most circumstances development teams are not disjoint: it is usual for a developer to be part of the teams allocated to four or five microservices. Having microservices written in different languages limit the ability to allocate available developers since it is not realistic to ask developers to wear the Java hat in the morning when they work on microservice A and wear the Python hat in the afternoon when allocated to service B. The same applies to different software development practices like pair programming, code reviews, coding styles and so forth. In this sense the human factor is what, in practice, dooms most dreams of decentralized governance.

Data management can be impacted when consistency requirements need some kind of transactional support, de facto excluding most NoSQL databases. However, as previously discussed, a single microservices application can adopt different consistency levels, thus different data management solutions can indeed be mixed but great care has to be taken to correctly identify whether a microservice is involved or not in requests with stringent consistency requirements.

As for platform/infrastructure, considerations similar to the ones expressed in respect to language can be proposed: it is not reasonable to think that each microservice reinvents the wheel and that infrastructure services are always written from scratch. The decision to embrace a specific event-based framework rather than a specific message broker, however, is not usually made (or at least it should not, to avoid the risk of having to re-think that decision later) simply because of a preference in the programming model or languages supported but first and foremost because of the guarantees that this solution provides in terms of non-functional dimensions.

The adoption of a messaging middleware to support asynchronous message-based solutions also impacts language: while database access drivers are usually available for a plethora of languages, messaging solutions are often restricted to a few languages (or even one).

Another impact of infrastructural decisions is due to the out-of-process mitigation strategies exposed in Sect. 3, from the large impact due to the adoption of a service mesh to the minor, but diffused impact of monitoring and logging solutions.

To summarize our analysis we could say that the design space for microservice architectures is indeed a large one. However, as soon as we start introducing strategies to improve availability and/or consistency, this space shrinks. This outcome is not unexpected since we know that non-functional requirements are the main driver behind software architectures.

The last part of this section deals with another aspect related to design space dimensions: the social one. In the social network era it should be no surprise that software developers tend to form an opinion on technological matters by reading IT social media.

But today's IT social media is flooded with narrations of microservices being the one solution that can bring freedom in software development when developing critical distributed applications, shadowing many of the complex issues that these systems unavoidably embody.

The scientific community is severely lagging behind in this evolution, while it is true that most of the basic mechanisms in what are proposed as modern solutions to these issues have been well known for dozens of years, it is also true that mixing the same ingredients in different doses and with different seasonings can result in a completely different experience. This results in insufficient support to practitioners who are left with no authoritative references when gathering information meant to inform architectural decisions.

Many of the solutions presented in this paper are complex. All this complexity should be no surprise to anyone with a strong background on distributed systems. The real issue is the number of developers with no, or very limited, distributed systems background who joined the microservices bandwagon, fascinated by a narration of freedom and complexity-free solutions. They design their systems without really understanding all the intricacies of a distributed system and they do not realize about the problems until they are in production. Finally, they understand why the very definition of an architectural error is an error that is costly to fix.

5 Conclusions

Microservice architectures can be built on top of very diverse foundations: different languages, different data management solutions, different interaction patterns and so forth. From a software developer point of view this results in a large decision space, allowing the design of applications able to meet a large spectrum of non-functional requirements (summarized in this paper with consistency and availability). A peculiar characteristic of microservices is the ability to adjust the availability/consistency slider on a service-by-service or even request-by-request basis, something that is possible also with different approaches but paying a price in terms of internal software qualities. Unexpectedly, as soon as we start to improve availability and/or consistency our design space dimensions are more and more constrained. Moreover, complexity creeps in, from both a software design and a system maintenance point of view.

All modern software development methods underline the importance of a risk-driven approach [12]: critical decisions should be taken as soon as possible during the development process in order to avoid the need to reconsider them later, which is not just costly but is something that can lead the whole project to a failure.

Developers adopting microservices should be very aware of that: the idea of starting with “something that works” and later “add on top availability and/or consistency” is always wrong. With microservices it is even worse. In practical terms this means that the first thing to do in a microservices-based software project is to clarify the needs in terms of non-functional requirements, decide the strategies to adopt to meet the requirements, understand how the design space changes on the basis of this adoption and pick a solution that is compatible with this design space.

At the end of the day non-functional requirements shape a software architecture. It has been like that for the whole history of software engineering, it is not going to change with microservices.

Acknowledgements. The work presented in this paper was partially supported by the MIUR PRIN 2015 GAUSS Project.

Appendix

Digression 1: The orchestrator is a microservice, which translates in yet another development project to maintain. Even with the minimal overhead imposed by agile methods the explosion of the number of projects can be troublesome. For this reason (and to improve time-to-market) several approaches based on configurable microservices orchestrators are starting to appear, in some sense we are witnessing the (dreaded) orchestration middleware from E-SOA making its appearance in disguise in the microservices world. Examples include Netflix's Conductor[4] (which uses a proprietary DSL) and Zeebe[5] (which uses BPMN).

Digression 2: the indirection caused by a messaging infrastructure in often mistaken by logical decoupling. It is true that with these solutions you could, for example, run each component in isolation and that changes in a component providing a function through messaging to another does not imply a change to the latter (which is an usual definition of dependency) but still a malfunction is going to happen so a form of coupling is present. This is because messaging based solution imply a form of hidden interfaces in which a logical dependency still holds between components but interfaces cannot be used as contracts to certify them. This could be summarized in the observation that message-based solutions are more flexible but the price to pay for that is maintenance.

Digression 3: enhancing existing configurable orchestrators with long-running transaction support seems like a reasonable option. This would essentially result in BPEL for microservices. To the best of the author's knowledge no product able to do that is currently available (albeit Zeebe seems a good candidate), it will be interesting to see what the future holds in this respect.

References

1. Buschmann, F., Henney, K., Schimdt, D.: Pattern-Oriented Software Architecture. On patterns and Pattern Languages, vol. 5. Wiley, New York (2007)
2. Rossi, D., Poggi, F., Ciancarini, P.: Dynamic high-level requirements in self-adaptive systems. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pp. 128–137. ACM, New York, NY, USA (2018)
3. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes **17**, 40–52 (1992)
4. Lewis, F., Fowler, M.: Microservices. https://martinfowler.com/articles/microservices.html
5. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available partition-tolerant web services. SIGACT News. **33**, 51–59 (2002)
6. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: Distributed Systems: Concepts and Design. Pearson, Boston (2011)
7. Nygard, M.T.: Release It! Design and Deploy Production-Ready Software. Pragmatic Bookshelf, Raleigh (2018)

[4] https://github.com/Netflix/conductor.

[5] https://zeebe.io.

8. Burns, B., Oppenheimer, D.: Design patterns for container-based distributed systems. Presented at the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16) (2016)
9. Morgan, W.: What's a Service Mesh? And Why Do I Need One?. https://dzone.com/articles/whats-a-service-mesh-and-why-do-i-need-one
10. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, pp. 1–7 (2011)
11. Xie, C., et al.: Salt: combining ACID and BASE in a distributed database. In: OSDI, pp. 495–509 (2014)
12. Boehm, B.W.: Software risk management: principles and practices. IEEE Softw. **8**, 32–41 (1991)
13. Pritchett, D.: BASE: an acid alternative. Queue. **6**, 48–55 (2008). https://doi.org/10.1145/1394127.1394128
14. Vogels, W.: Eventually consistent. Commun. ACM. **52**, 40–44 (2009). https://doi.org/10.1145/1435417.1435432
15. Bailis, P., Fekete, A., Franklin, M.J., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Feral concurrency control: an empirical investigation of modern application integrity. Presented at the Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 27 May 2015. https://doi.org/10.1145/2723372.2737784
16. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, pp. 249–259. ACM, New York, NY, USA (1987). https://doi.org/10.1145/38713.38742