# Transparent cross-system consistency

João Loff
INESC-ID/IST – U. Lisboa

Daniel Porto
INESC-ID/IST – U. Lisboa

Carlos Baquero
HASLab/INESC-TEC – U. Minho
*

João Garcia
INESC-ID/IST – U. Lisboa

Nuno Preguiça
NOVALINCS/FCT – UNL

Rodrigo Rodrigues
INESC-ID/IST – U. Lisboa

## Abstract

This paper discusses the motivation and the challenges for providing a systematic and transparent approach for dealing with cross-system consistency. Our high level goal is to provide a way to avoid violations of causality when multiple systems interact, while (a) avoiding the redesign of existing systems, (b) minimizing the overhead, and (c) requiring as little developer input as possible.

***Keywords*** Consistency; Microservices; Causality; Coordination.

## 1 Introduction

Ever-growing user bases challenge Internet companies to design applications that meet or even exceed the the expectations of the end-user service experience. Keeping users engaged is important for business [11] and, conversely, unsatisfied users end up leaving the service, thus resulting in loss of revenue [16].

To address both performance and functionality requirements, the software infrastructure that underlies Internet services normally resorts to an architecture in which systems are split into multiple independent cooperating services, where each service can either be a full blown distributed system such as a mix of storage, processing, and monitoring systems [**?** ], or also smaller scale systems such as *microservices* [12]. This architecture is not only motivated by modularity, but it also makes it easier to individually optimize each system, e.g., in terms of performance and scalability.

However, scattering application data functionality across multiple services comes at a price. A single user request may span hundreds of sub-queries that traverse multiple different services [4], with possibly different consistency semantics. Consequently, copies of the same (or causally related) data are spread throughout multiple services, jeopardizing causal relationships. For instance, at Facebook, notifications and posts, although inter-dependent, are processed by independent systems [1]. Hence, and regardless of the consistency level guaranteed by individual systems, inconsistencies can be present in the overall system, and ultimately perceived by the end user [1, 3].

In fact, exposing inconsistencies to the end-user, may lead to a direct negative impact on the quality of the user experience, and is therefore a topic that is monitored by major Internet companies. Facebook, for example, saw only a small prevalence of inconsistencies in one of their storage systems [13], but is looking at a way to systematically solve these issues when they occur across different systems [1]. In particular, when notifications for new posts are delivered before those posts are available, users flock to community forums to complain about these causality violations [6–9].

A possible way to orchestrate the behavior of multiple interacting systems, in order to control the overal semantics perceived by end users is through the use of an external coordination mechanism such as Zookeeper [10]. While this is an effective way to prevent

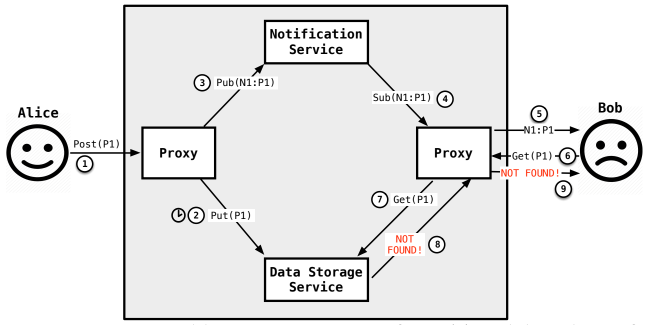


**Figure 1.** User visible inconsistencies. If e.g. (2) is delayed, notification can be delivered without posts being available.

anomalies from surfacing (as operations are only allowed to proceed under certain conditions), this solution has three important drawbacks. First, it requires the developer to identify these pitfalls, which requires deep understanding of the consistency models involved and their resulting interplay. Second, it requires significant modifications to the various systems in order to invoke this coordination in the appropriate moments. Finally, coordination systems like Zookeeper represent a serialization point in the system, which may introduce significant delays [2]. In fact, a mere fork/join structure of requests when a single sub-query needs coordination can cause a cascading slowdown throughout the whole system [1, 15].

Given these difficulties, we argue that it is important to find a solution addresses the problem of the complex and subtle interactions between ecosystems of multiple systems or micro-services, with various interfaces and semantics. Ideally, such a solution should meet the following requirements: (1) compatibility with existing systems, namely avoiding their redesign, (2) introducing only the necessary amount of coordination, thus minimizing the coordination overhead, (3) minimizing the burden on the developers, by requiring only a small amount of annotations to existing code bases, and without requiring an understanding of the subtle ways in which the semantics of the different systems may interact, and (4) preventing non-intuitive behaviors from being exposed to end users.

The remainder of this paper provides an overview of the challenges for providing such an approach, and the key techniques that we can leverage as building blocks.

## 2 Challenges in cross-system causality

As an illustrative example, consider a simple Facebook-like *reactive* application composed by a *Proxy*, a *Data Store* and a *Notification Service*. *Proxies* handle client requests, such as posting a new message. When handling this type of request, the proxy stores the new post on the *Data Store*, and publishes a notification to the *Notification Service*. The problem that may arise in this scenario is that, even if each independent service has strong consistency guarantees (such as linearizability), consistency violations can happen [17]. For instance, as shown in Figure 1, after processing the request from alice ($\overrightarrow{1}$), the Proxy splits her post into two different objects: Content ($\overrightarrow{2}$) and Notification ($\overrightarrow{3}$). Although both objects are causally related, it is possible that one object is delivered ($\overrightarrow{3,4,5}$) while the other ($\overrightarrow{2}$) could be delayed (or even aborted). As a result, after seeing a notification for the new post ($\overrightarrow{5}$), Bob may try to access the post ($\overrightarrow{6,7}$) and, because its content is not yet available ($\overrightarrow{2}$), Bob ends up receiving an error message as a response ($\overrightarrow{8,9}$).

A simple solution with a coordination mechanism can be employed to prevent this problem from happening. As shown in Figure 2a, the notification for Alice's post can be propagated to the *Notification Service* ($\overrightarrow{1,2,3}$), but, before being delivered, the application logic must use a coordination service in order to wait ($\overrightarrow{4}$) until the post content becomes available in the *Data Store* ($\overrightarrow{1,2,5}$). Only then, the coordinator allows the notification to be delivered to Bob ($\overrightarrow{6,7}$). Finally, when Bob visualizes the notification from the new post that Alice issued and tries to access it ($\overrightarrow{8,9}$), the post content is now available and can be returned to Bob.

Note that this solution has two drawbacks. First, both services have to adhere to a common coordination protocol. Second, two additional round trip messages are required to make sure that the notification can be propagated safely to the *Proxy* ($\overrightarrow{4}$ and $\overrightarrow{5}$), which introduce delays. Alternatively, the *Notification* and *Data Store* services could be modified to coordinate directly among themselves and save a round trip. Nevertheless, the refactoring effort of these services may be very high compared to centralizing that logic in the application code.

The problem that is illustrated by this simple example may be exacerbated in real systems. This is because, over time, Internet applications grow in features, and thus need to connect different services to provide a richer experience. For instance, the microservices definition advocates application decomposition based on the principles of replacement and upgradeability, favoring a granular release planning [12]. Hence, new features are added by implementing new microservices, thus resulting in a system design that is composed by a large number of inter-dependent microservices. For instance, Netflix is reported to have hundreds of microservices [4]. Consequently, coordination amongst a growing number of services amplifies that above mentioned challenges.

## 3 Towards cross-system consistency

Figure 2b depicts the blueprint for our proposal. The shaded rectangles interposed between existing components form our proposed transparent coordination layer. At a high level, our coordination layer intercepts cross-system communication, adding metadata to each request. This metadata allows us to keep track of data dependencies and prevent cross-system consistency violations. Next, we outline the goals and strategies for this layer.

*Avoid changes in current systems.* Ideally, we want to avoid changing existing consistency and event-ordering mechanisms, while averting the drawbacks discussed in §2. Hence, instead of changing the design of existing systems, we aim to augment existing cross-system communication interfaces so they transparently provide consistency between subsystems. In particular, by instrumenting existing systems to intercept common cross-system communication APIs (such as RPC frameworks), it is possible to intercept the metadata appended to the messages to validate causal dependencies without affecting the execution of the system. Note that this strategy was previously adopted in the context of other problem statements. Crane [5], for example, uses a similar strategy to achieve transparent state machine replication by changing the common socket API.

*Minimize coordination requirements.* By leveraging the existing *baggage* abstraction [14], we intend to derive at runtime *happened-before* relations between cross-systems operations. This can enable us to build *causal histories* for each request, which, in turn, allow us to know, in any point of the execution, what is the chain of operations that led to that point.

*Minimize developer input.* The two components we described (intercepting communication APIs and baggages) already have the potential to enable capturing the full causal history of requests, with a small amount of developer effort. In addition to these two components, another key aspect that our approach enables is that, by comparing the causal histories of various operations on different systems, we are able to suspect causality violations without requiring the developer to reason about the consistency definitions of individual systems, which are hard to understand per se, and the resulting semantics when these systems are interconnected.

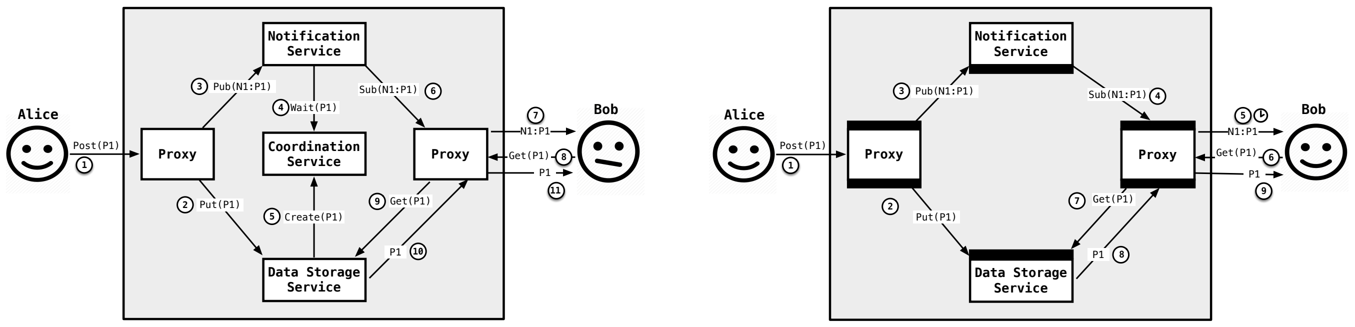
*Validating cross-system requests.* Thanks to the use of baggages, we are closer to our goal of being able to detect the following scenario: there is an event B (e.g., a notification) that depends on another event A that happened before B (e.g., a post). Then B triggers an event D (looking up a value in cache), where the state of the cache is such that it should have seen the effects of A but it didn't. In particular, event D is associated with a baggage that reflects the dependency on event A, but the baggage that is associated with the cache state does not reflect the dependency on A.

The challenge, however, is that there is also the possibility that the cache will never be affected by A, and therefore the fact that the baggage associated with the cache state does not refect A is not a violation of causality.

To address this, we envision that the system goes through a training phase, which determines automatically the fact that the cache system is normally supposed to reflect posts. This training phase builds all the dependence graphs that the system generates using baggages, and builds templates associated with typical fork-join patterns such as the one described.

*Preventing anomalous behaviors.* When a request is matched against one of the invalid templates, we prevent anomalous behavior by simply intercepting requests (at our instrumentation layer level), and delaying their delivery until some happened-before condition is met. In other words, we delay requests until some other event on another system has completed, deeming the current request's causal history no longer invalid (and hence it can be safely delivered).

This raises another challenge, illustrated by the example in Figure 2b, which is that there are two ways to achieve this. The first one is to delay the execution of reading data from the cache ($\overrightarrow{7}$) until its causal dependency is met, but this is undesirable since it will cause the user to notice this delay. As such, we would prefer a solution where we prevent the notification from being delivered to



**(a)** Coordination service. Consistency plus extra communication steps.

**(b)** Transparent coordination. Prevents causality violations.

**Figure 2.** Coordination strategies

users ($\overrightarrow{5}$) before making sure that posts are stored ($\overrightarrow{2}$). Consider ($\overrightarrow{2}$) *happens-before* ($\overrightarrow{5}$), we can proactively delay the delivery of ($\overrightarrow{5}$) until we know ($\overrightarrow{2}$) has been completed.

Note that, while we were discussing ($\overrightarrow{5}$), we could instead make the exact same case for ($\overrightarrow{4}$). We could delay ($\overrightarrow{4}$) until the Notification Service has seen ($\overrightarrow{2}$). Would a developer rather prevent a notification to be delivered to the proxy or to the client? There is no correct answer, since this choice is applications specific. Therefore this action requires developer input.

## 4 Final remarks

In this position paper we discussed the challenges for providing a systematic and transparent approach for dealing with cross-system consistency.

## References

[1] Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*. https://www.usenix.org/conference/hotos15/workshop-program/presentation/ajoux

[2] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *VLDB Endowment* 8, 3 (nov 2014), 185–196. DOI: http://dx.doi.org/10.14778/2735508.2735509

[3] David R. Cheriton and Dale Skeen. 1993. Understanding the limitations of causally and totally ordered communication. *ACM SIGOPS Operating Systems* 27, 5 (dec 1993), 44–57. DOI: http://dx.doi.org/10.1145/173668.168623

[4] Adrian Cockcroft. 2014. Migrating to Cloud Native with Microservices. In *GOTO Conference (GOTOCon '14)*. 76. http://gotocon.com/dl/goto-berlin-2014/slides/AdrianCockcroft

[5] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. Paxos made transparent. *25th Symposium on Operating Systems Principles (SOSP '15)* (2015), 105–120. DOI: http://dx.doi.org/10.1145/2815400.2815427

[6] Facebook Help Community. 2017. Anyone know why I can click on a post and I get the page not found? (2017). https://www.facebook.com/help/community/question/?id=1062960447061148

[7] Facebook Help Community. 2017. Notification links with picture only brings to page not found. (2017).

[8] Facebook Help Community. 2017. Why am i Not receiving all of my notifications on posts that i comment on? (2017). https://www.facebook.com/help/community/question/?id=1514215372130647

[9] Facebook Help Community. 2017. Why when I get notifications but then not showing up on my page? (2017). https://www.facebook.com/help/community/question/?id=10152452521000351

[10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. *USENIX Annual Technical Conference* 8 (2010), 9. http://portal.acm.org/citation.cfm?id=1855851

[11] Sanjeev Kulkarni, Nikunj Bhagat, Masong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron. *ACM SIGMOD International Conference on Management of Data (SIGMOD '15)* (2015), 239–250. DOI: http://dx.doi.org/10.1145/2723372.2742788

[12] J. Lewis and M. Fowler. 2016. Microservices: A definition of this new architectural term. (2016). https://martinfowler.com/articles/microservices.html

[13] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency. In *25th Symposium on Operating Systems Principles (SOSP '15)*. ACM Press, New York, New York, USA, 295–310. DOI: http://dx.doi.org/10.1145/2815400.2815426

[14] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic causal monitoring for distributed systems. In *Symposium on Operating Systems Principles (SOSP '15)*. 378–393. DOI: http://dx.doi.org/10.1145/2815400.2815415

[15] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal!. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi

[16] E Schurman and J Brutlag. 2009. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. (2009). http://scholar.google.com/scholar?hl=en

[17] Irene Zhang, Niel Lebeck, Ariadna Norberg, Pedro Fonseca, Brandon Holt, Raymond Cheng, Arvind Krishnamurthy, and Henry M Levy. 2016. Diamond: Automating Data Management and Storage for Wide-area, Reactive Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 723–738.